The transport layer

As the transport layer is built on top of the network layer, it is important to know the key features of the network layer service. There are two types of network layer services : connectionless and connection-oriented. The connectionless network layer service is the most widespread. Its main characteristics are :

- the connectionless network layer service can only transfer SDUs of *limited size* [1]
- the connectionless network layer service may discard SDUs
- the connectionless network layer service may corrupt SDUs
- the connectionless network layer service may delay, reorder or even duplicate SDUs

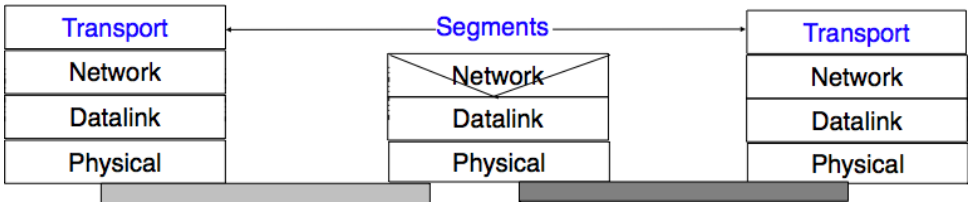

Figure 4.1: The transport layer in the reference model

These imperfections of the connectionless network layer service will become much clearer once we have explained the network layer in the next chapter. At this point, let us simply assume that these imperfections occur without trying to understand why they occur.

Some transport protocols can be used on top of a connection-oriented network service, such as class 0 of the ISO Transport Protocol (TP0) defined in [X224] , but they have not been widely used. We do not discuss in further detail such utilisation of a connection-oriented network service in this book.

This chapter is organised as follows. We will first explain how it is possible to provide a reliable transport service on top of an unreliable connectionless network service. For this, we explain the main mechanisms found in such protocols. Then, we will study in detail the two transport protocols that are used in the Internet. We begin with the User Datagram Protocol (UDP) which provides a simple connectionless transport service. Then, we will describe in detail the Transmission Control Protocol (TCP), including its congestion control mechanism.

4.1 Principles of a reliable transport protocol

In this section, we depict a reliable transport protocol running above a connectionless network layer service. For this, we first assume that the network layer provides a perfect service, i.e. :

- the connectionless network layer service never corrupts SDUs

[1] Many network layer services are unable to carry SDUs that are larger than 64 KBytes.

- the connectionless network layer service never discards SDUs
- the connectionless network layer service never delays, reorders nor duplicate SDUs
- the connectionless network layer service can support SDUs of *any size*

We will then remove each of these assumptions one after the other in order to better understand the mechanisms used to solve each imperfection.

4.1.1 Reliable data transfer on top of a perfect network service

The transport layer entity interacts with both a user in the application layer and an entity in the network layer. According to the reference model, these interactions will be performed using *DATA.req* and *DATA.ind* primitives. However, to simplify the presentation and to avoid confusion between a *DATA.req* primitive issued by the user of the transport layer entity, and a *DATA.req* issued by the transport layer entity itself, we will use the following terminology :

- the interactions between the user and the transport layer entity are represented by using the classical *DATA.req*, *DATA.ind* primitives
- the interactions between the transport layer entity and the network layer service are represented by using *send* instead of *DATA.req* and *recvd* instead of *DATA.ind*

This is illustrated in the figure below.

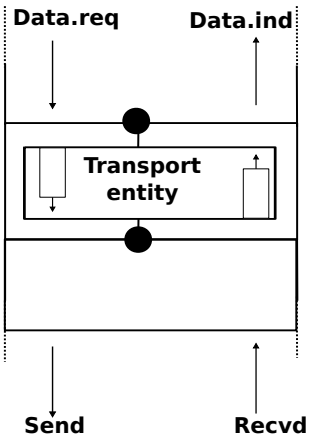

Figure 4.2: Interactions between the transport layer, its user, and its network layer provider

When running on top of a perfect connectionless network service, a transport level entity can simply issue a *send(SDU)* upon arrival of a *DATA.req(SDU)*. Similarly, the receiver issues a *DATA.ind(SDU)* upon receipt of a *recvd(SDU)*. Such a simple protocol is sufficient when a single SDU is sent.

Unfortunately, this is not always sufficient to ensure a reliable delivery of the SDUs. Consider the case where a client sends tens of SDUs to a server. If the server is faster that the client, it will be able to receive and process all the segments sent by the client and deliver their content to its user. However, if the server is slower than the client, problems may arise. The transport layer entity contains buffers to store SDUs that have been received as a *Data.request* from the application but have not yet been sent via the network service. If the application is faster than the network layer, the buffer becomes full and the operating system suspends the application to let the transport entity empty its transmission queue. The transport entity also uses a buffer to store the segments received from the network layer that have not yet been processed by the application. If the application is slow to process the data, this buffer becomes full and the transport entity is not able to accept anymore the segments from the network layer. The buffers of the transport entity have a limited size [2] and if they overflow, the transport entity is forced to

[2] In the application layer, most servers are implemented as processes. The network and transport layer on the other hand are usually implemented inside the operating system and the amount of memory that they can use is limited by the amount of memory allocated to the entire kernel.

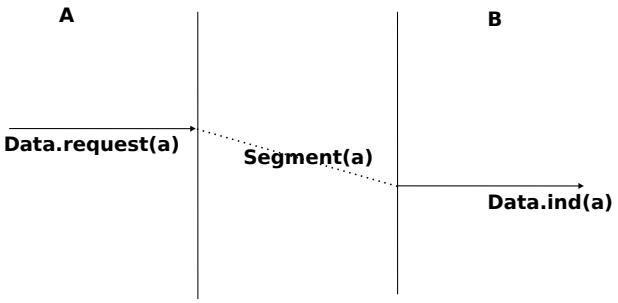

Figure 4.3: The simplest transport protocol

discard received segments.

To solve this problem, our transport protocol must include a feedback mechanism that allows the receiver to inform the sender that it has processed a segment and that another one can be sent. This feedback is required even though the network layer provides a perfect service. To include such a feedback, our transport protocol must process two types of segments :

- data segments carrying a SDU
- control segments carrying an acknowledgment indicating that the previous segment was processed correctly

These two types of segments can be distinguished using a segment composed of two parts :

- the *header* that contains one bit set to *0* in data segments and set to *1* in control segments
- the payload that contains the SDU supplied by the user application

The transport entity can then be modelled as a finite state machine, containing two states for the receiver and two states for the sender. The figure below provides a graphical representation of this state machine with the sender above and the receiver below.

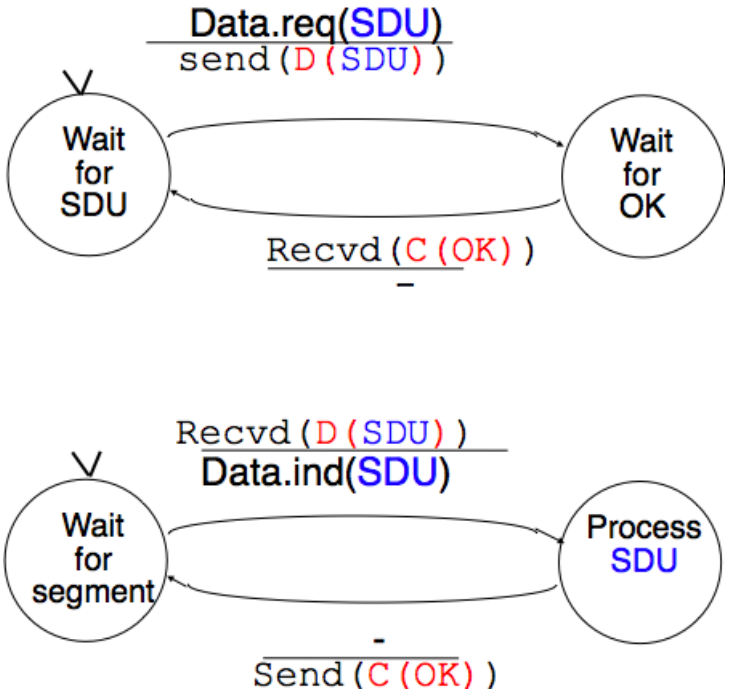

Figure 4.4: Finite state machine of the simplest transport protocol

The above FSM shows that the sender has to wait for an acknowledgement from the receiver before being able to transmit the next SDU. The figure below illustrates the exchange of a few segments between two hosts.

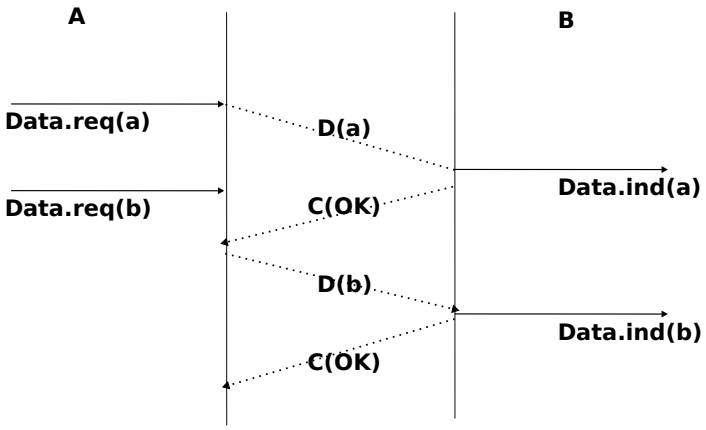

Figure 4.5: Time sequence diagram illustrating the operation of the simplest transport protocol

4.1.2 Reliable data transfer on top of an imperfect network service

The transport layer must deal with the imperfections of the network layer service. There are three types of imperfections that must be considered by the transport layer :

1. Segments can be corrupted by transmission errors
2. Segments can be lost
3. Segments can be reordered or duplicated

To deal with these types of imperfections, transport protocols rely on different types of mechanisms. The first problem is transmission errors. The segments sent by a transport entity is processed by the network and datalink layers and finally transmitted by the physical layer. All of these layers are imperfect. For example, the physical layer may be affected by different types of errors :

- random isolated errors where the value of a single bit has been modified due to a transmission error
- random burst errors where the values of *n* consecutive bits have been changed due to transmission errors
- random bit creations and random bit removals where bits have been added or removed due to transmission errors

The only solution to protect against transmission errors is to add redundancy to the segments that are sent. *Information Theory* defines two mechanisms that can be used to transmit information over a transmission channel affected by random errors. These two mechanisms add redundancy to the information sent, to allow the receiver to detect or sometimes even correct transmission errors. A detailed discussion of these mechanisms is outside the scope of this chapter, but it is useful to consider a simple mechanism to understand its operation and its limitations.

Information theory defines *coding schemes*. There are different types of coding schemes, but let us focus on coding schemes that operate on binary strings. A coding scheme is a function that maps information encoded as a string of *m* bits into a string of *n* bits. The simplest coding scheme is the even parity coding. This coding scheme takes an *m* bits source string and produces an *m+1* bits coded string where the first *m* bits of the coded string are the bits of the source string and the last bit of the coded string is chosen such that the coded string will always contain an even number of bits set to *1*. For example :

- *1001* is encoded as *10010*
- *1101* is encoded as *11011*

This parity scheme has been used in some RAMs as well as to encode characters sent over a serial line. It is easy to show that this coding scheme allows the receiver to detect a single transmission error, but it cannot correct it. However, if two or more bits are in error, the receiver may not always be able to detect the error.

Some coding schemes allow the receiver to correct some transmission errors. For example, consider the coding scheme that encodes each source bit as follows :

- *1* is encoded as *111*
- *0* is encoded as *000*

For example, consider a sender that sends *111*. If there is one bit in error, the receiver could receive *011* or *101* or *110*. In these three cases, the receiver will decode the received bit pattern as a *1* since it contains a majority of bits set to *1*. If there are two bits in error, the receiver will not be able anymore to recover from the transmission error.

This simple coding scheme forces the sender to transmit three bits for each source bit. However, it allows the receiver to correct single bit errors. More advanced coding systems that allow to recover from errors are used in several types of physical layers.

Transport protocols use error detection schemes, but none of the widely used transport protocols rely on error correction schemes. To detect errors, a segment is usually divided into two parts :

- a *header* that contains the fields used by the transport protocol to ensure reliable delivery. The header contains a checksum or Cyclical Redundancy Check (CRC) [Williams1993] that is used to detect transmission errors
- a *payload* that contains the user data passed by the application layer.

Some segment headers also include a *length* , which indicates the total length of the segment or the length of the payload.

The simplest error detection scheme is the checksum. A checksum is basically an arithmetic sum of all the bytes that a segment is composed of. There are different types of checksums. For example, an eight bit checksum can be computed as the arithmetic sum of all the bytes of (both the header and trailer of) the segment. The checksum is computed by the sender before sending the segment and the receiver verifies the checksum upon reception of each segment. The receiver discards segments received with an invalid checksum. Checksums can be easily implemented in software, but their error detection capabilities are limited. Cyclical Redundancy Checks (CRC) have better error detection capabilities [SGP98], but require more CPU when implemented in software.

Note: Checksums, CRCs, ...

Most of the protocols in the TCP/IP protocol suite rely on the simple Internet checksum in order to verify that the received segment has not been affected by transmission errors. Despite its popularity and ease of implementation, the Internet checksum is not the only available checksum mechanism. Cyclical Redundancy Checks (CRC) are very powerful error detection schemes that are used notably on disks, by many datalink layer protocols and file formats such as zip or png. They can easily be implemented efficiently in hardware and have better error-detection capabilities than the Internet checksum [SGP98] . However, when the first transport protocols were designed, CRCs were considered to be too CPU-intensive for software implementations and other checksum mechanisms were used instead. The TCP/IP community chose the Internet checksum, the OSI community chose the Fletcher checksum [Sklower89] . Now, there are efficient techniques to quickly compute CRCs in software [Feldmeier95] , the SCTP protocol initially chose the Adler-32 checksum but replaced it recently with a CRC (see **RFC 3309**).

The second imperfection of the network layer is that segments may be lost. As we will see later, the main cause of packet losses in the network layer is the lack of buffers in intermediate routers. Since the receiver sends an acknowledgement segment after having received each data segment, the simplest solution to deal with losses is to use a retransmission timer. When the sender sends a segment, it starts a retransmission timer. The value of this retransmission timer should be larger than the *round-trip-time*, i.e. the delay between the transmission of a data segment and the reception of the corresponding acknowledgement. When the retransmission timer expires, the sender assumes that the data segment has been lost and retransmits it. This is illustrated in the figure below.

Unfortunately, retransmission timers alone are not sufficient to recover from segment losses. Let us consider, as an example, the situation depicted below where an acknowledgement is lost. In this case, the sender retransmits

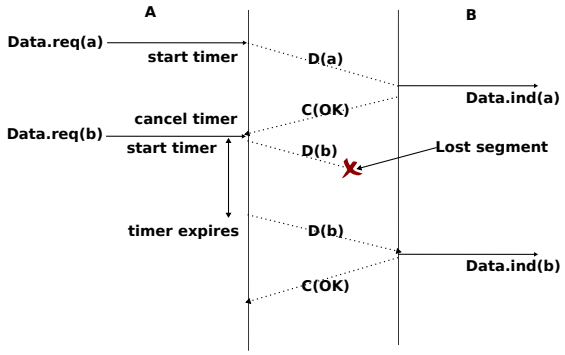

Figure 4.6: Using retransmission timers to recover from segment losses

the data segment that has not been acknowledged. Unfortunately, as illustrated in the figure below, the receiver considers the retransmission as a new segment whose payload must be delivered to its user.

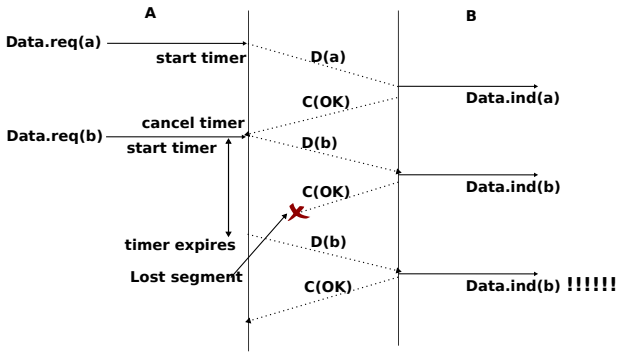

Figure 4.7: Limitations of retransmission timers

To solve this problem, transport protocols associate a *sequence number* to each data segment. This *sequence number* is one of the fields found in the header of data segments. We use the notation *D(S,...)* to indicate a data segment whose sequence number field is set to *S*. The acknowledgements also contain a sequence number indicating the data segments that it is acknowledging. We use *OKS* to indicate an acknowledgement segment that confirms the reception of *D(S,...)*. The sequence number is encoded as a bit string of fixed length. The simplest transport protocol is the Alternating Bit Protocol (ABP).

The Alternating Bit Protocol uses a single bit to encode the sequence number. It can be implemented easily. The sender and the receivers only require a four states Finite State Machine.

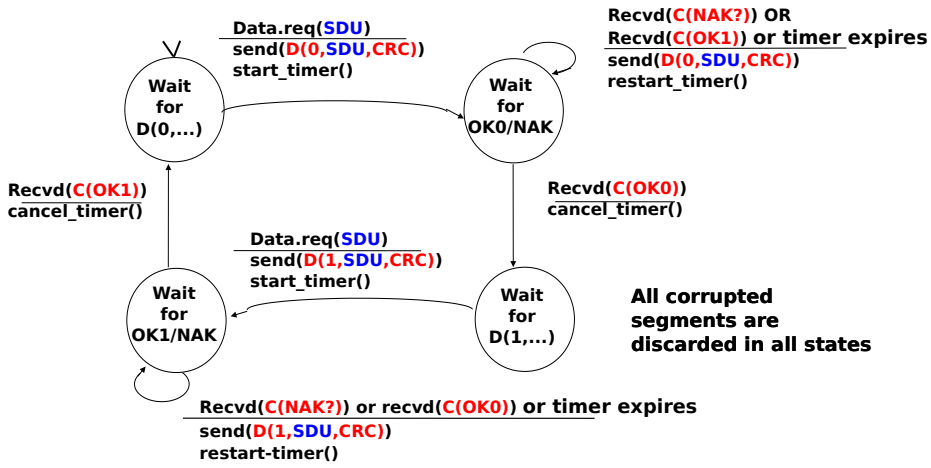

Figure 4.8: Alternating bit protocol : Sender FSM

The initial state of the sender is *Wait for D(0,...)*. In this state, the sender waits for a *Data.request*. The first data segment that it sends uses sequence number *0*. After having sent this segment, the sender waits for an *OK0* acknowledgement. A segment is retransmitted upon expiration of the retransmission timer or if an acknowledgement with an incorrect sequence number has been received.

The receiver first waits for *D(0,...)*. If the segment contains a correct *CRC*, it passes the SDU to its user and sends *OK0*. If the segment contains an invalid CRC, it is immediately discarded. Then, the receiver waits for *D(1,...)*. In this state, it may receive a duplicate *D(0,...)* or a data segment with an invalid CRC. In both cases, it returns an *OK0* segment to allow the sender to recover from the possible loss of the previous *OK0* segment.

Note: Dealing with corrupted segments

The receiver FSM of the Alternating bit protocol discards all segments that contain an invalid CRC. This is the safest approach since the received segment can be completely different from the segment sent by the remote host. A receiver should not attempt at extracting information from a corrupted segment because it cannot know which portion of the segment has been affected by the error.

The figure below illustrates the operation of the alternating bit protocol.

The Alternating Bit Protocol can recover from transmission errors and segment losses. However, it has one important drawback. Consider two hosts that are directly connected by a 50 Kbits/sec satellite link that has a 250 milliseconds propagation delay. If these hosts send 1000 bits segments, then the maximum throughput that can be achieved by the alternating bit protocol is one segment every $20 + 250 + 250 = 520$ milliseconds if we ignore the transmission time of the acknowledgement. This is less than 2 Kbits/sec !

Go-back-n and selective repeat

To overcome the performance limitations of the alternating bit protocol, transport protocols rely on *pipelining*. This technique allows a sender to transmit several consecutive segments without being forced to wait for an acknowledgement after each segment. Each data segment contains a sequence number encoded in an n bits field.

Pipelining allows the sender to transmit segments at a higher rate, but we need to ensure that the receiver does not

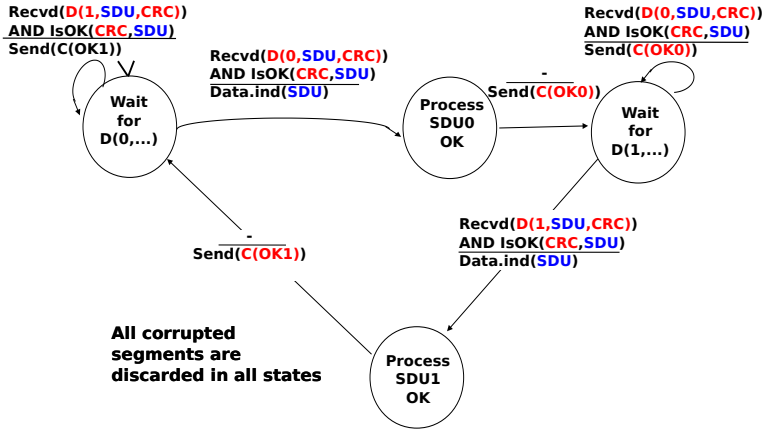

Figure 4.9: Alternating bit protocol : Receiver FSM

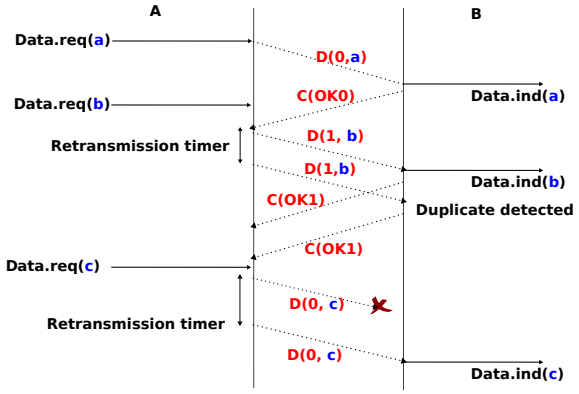

Figure 4.10: Operation of the alternating bit protocol

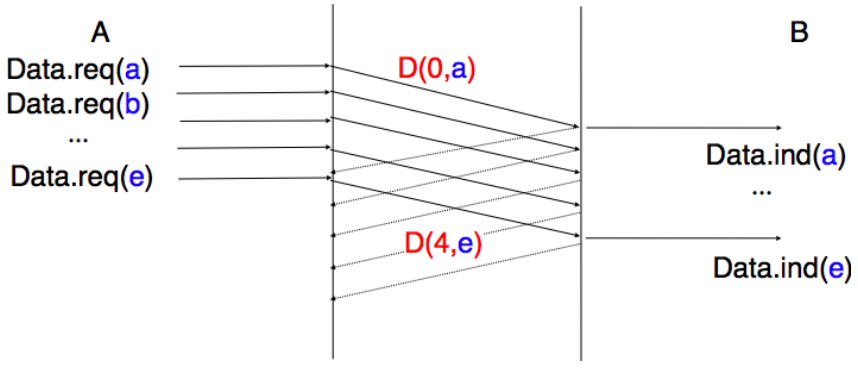

Figure 4.11: Pipelining to improve the performance of transport protocols

become overloaded. Otherwise, the segments sent by the sender are not correctly received by the destination. The transport protocols that rely on pipelining allow the sender to transmit *W* unacknowledged segments before being forced to wait for an acknowledgement from the receiving entity.

This is implemented by using a *sliding window*. The sliding window is the set of consecutive sequence numbers that the sender can use when transmitting segments without being forced to wait for an acknowledgement. The figure below shows a sliding window containing five segments (*6,7,8,9* and *10*). Two of these sequence numbers (*6* and *7*) have been used to send segments and only three sequence numbers (*8*, *9* and *10*) remain in the sliding window. The sliding window is said to be closed once all sequence numbers contained in the sliding window have been used.

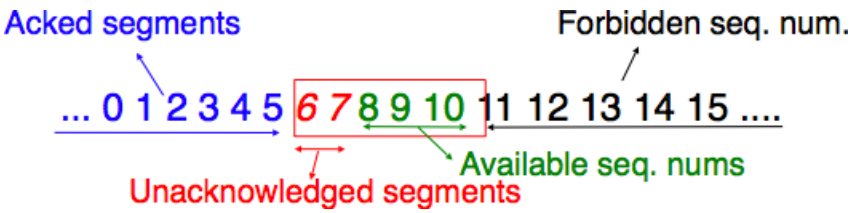

Figure 4.12: The sliding window

The figure below illustrates the operation of the sliding window. The sliding window shown contains three segments. The sender can thus transmit three segments before being forced to wait for an acknowledgement. The sliding window moves to the higher sequence numbers upon reception of acknowledgements. When the first acknowledgement (*OK0*) is received, it allows the sender to move its sliding window to the right and sequence number *3* becomes available. This sequence number is used later to transmit SDU *d*.

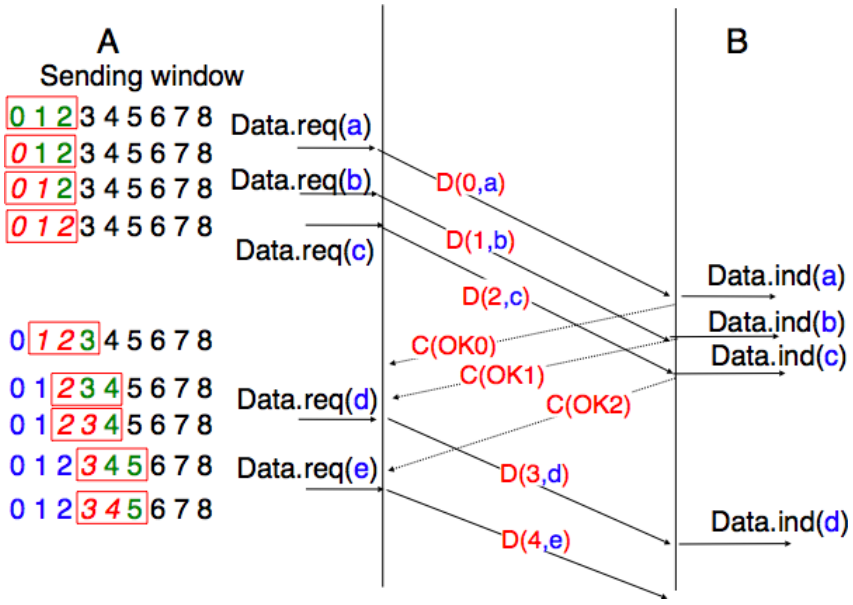

Figure 4.13: Sliding window example

In practice, as the segment header encodes the sequence number in an *n* bits string, only the sequence numbers between 0 and $2^n - 1$ can be used. This implies that the same sequence number is used for different segments and that the sliding window will wrap. This is illustrated in the figure below assuming that *2* bits are used to encode the sequence number in the segment header. Note that upon reception of *OK1*, the sender slides its window and can use sequence number *0* again.

Unfortunately, segment losses do not disappear because a transport protocol is using a sliding window. To recover from segment losses, a sliding window protocol must define :

- a heuristic to detect segment losses
- a *retransmission strategy* to retransmit the lost segments.

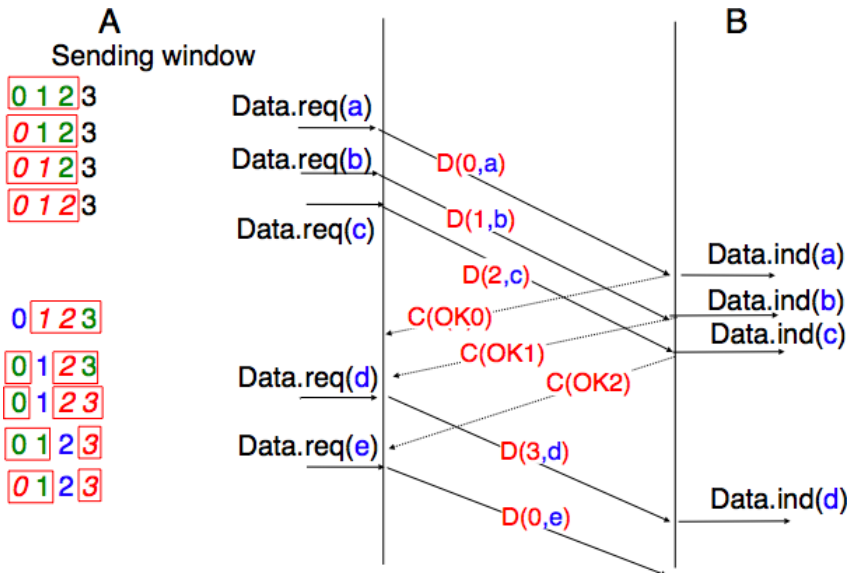

Figure 4.14: Utilisation of the sliding window with modulo arithmetic

The simplest sliding window protocol uses *go-back-n* recovery. Intuitively, *go-back-n* operates as follows. A *go-back-n* receiver is as simple as possible. It only accepts the segments that arrive in-sequence. A *go-back-n* receiver discards any out-of-sequence segment that it receives. When *go-back-n* receives a data segment, it always returns an acknowledgement containing the sequence number of the last in-sequence segment that it has received. This acknowledgement is said to be *cumulative*. When a *go-back-n* receiver sends an acknowledgement for sequence number *x*, it implicitly acknowledges the reception of all segments whose sequence number is earlier than *x*. A key advantage of these cumulative acknowledgements is that it is easy to recover from the loss of an acknowledgement. Consider for example a *go-back-n* receiver that received segments *1*, *2* and *3*. It sent *OK1*, *OK2* and *OK3*. Unfortunately, *OK1* and *OK2* were lost. Thanks to the cumulative acknowledgements, when the receiver receives *OK3*, it knows that all three segments have been correctly received.

The figure below shows the FSM of a simple *go-back-n* receiver. This receiver uses two variables : *lastack* and *next*. *next* is the next expected sequence number and *lastack* the sequence number of the last data segment that has been acknowledged. The receiver only accepts the segments that are received in sequence. *maxseq* is the number of different sequence numbers (2^n).

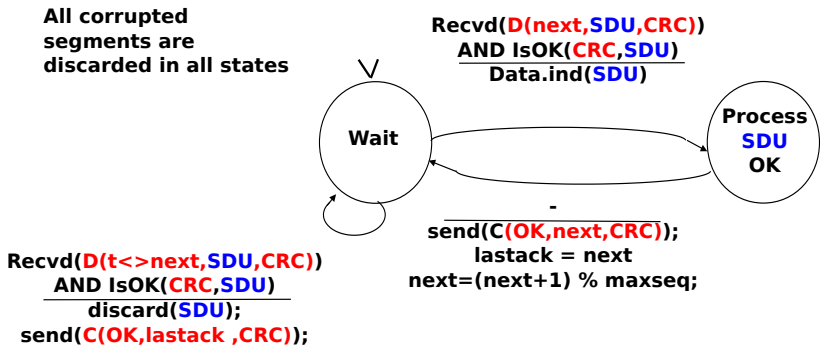

Figure 4.15: Go-back-n : receiver FSM

A *go-back-n* sender is also very simple. It uses a sending buffer that can store an entire sliding window of segments [3] . The segments are sent with increasing sequence number (modulo *maxseq*). The sender must wait for

[3] The size of the sliding window can be either fixed for a given protocol or negotiated during the connection establishment phase. We'll see later that it is also possible to change the size of the sliding window during the connection's lifetime.

an acknowledgement once its sending buffer is full. When a *go-back-n* sender receives an acknowledgement, it removes from the sending buffer all the acknowledged segments and uses a retransmission timer to detect segment losses. A simple *go-back-n* sender maintains one retransmission timer per connection. This timer is started when the first segment is sent. When the *go-back-n sender* receives an acknowledgement, it restarts the retransmission timer only if there are still unacknowledged segments in its sending buffer. When the retransmission timer expires, the *go-back-n* sender assumes that all the unacknowledged segments currently stored in its sending buffer have been lost. It thus retransmits all the unacknowledged segments in the buffer and restarts its retransmission timer.

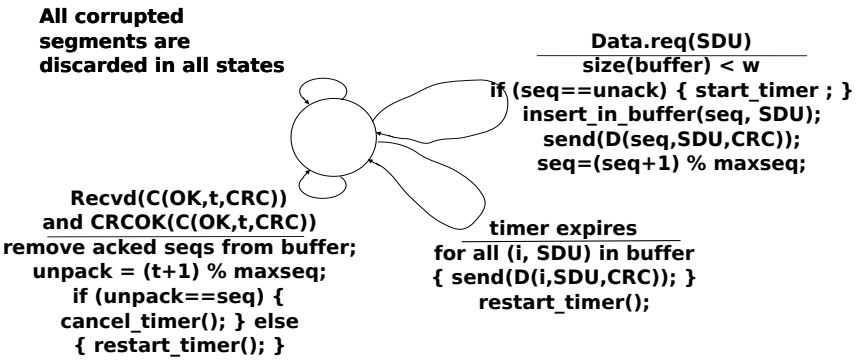

Figure 4.16: Go-back-n : sender FSM

The operation of *go-back-n* is illustrated in the figure below. In this figure, note that upon reception of the out-of-sequence segment *D(2,c)*, the receiver returns a cumulative acknowledgement *C(OK,0)* that acknowledges all the segments that have been received in sequence. The lost segment is retransmitted upon the expiration of the retransmission timer.

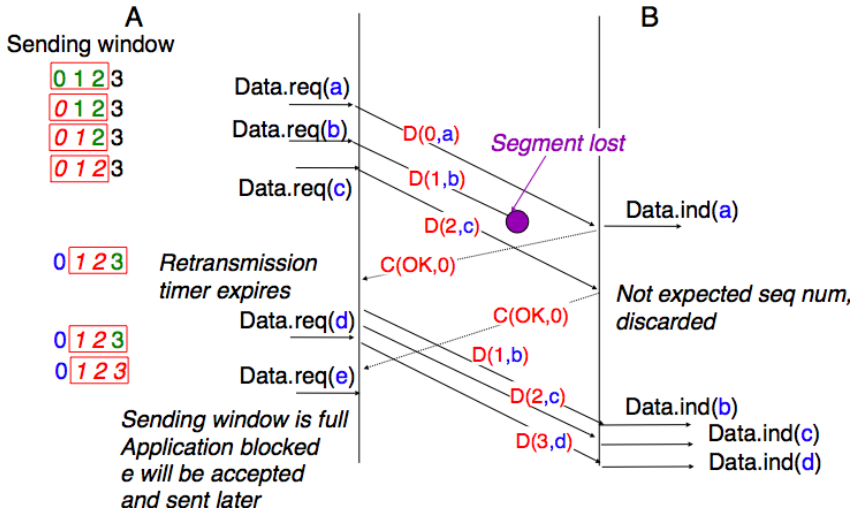

Figure 4.17: Go-back-n : example

The main advantage of *go-back-n* is that it can be easily implemented, and it can also provide good performance when only a few segments are lost. However, when there are many losses, the performance of *go-back-n* quickly drops for two reasons :

- the *go-back-n* receiver does not accept out-of-sequence segments
- the *go-back-n* sender retransmits all unacknowledged segments once its has detected a loss

Selective repeat is a better strategy to recover from segment losses. Intuitively, *selective repeat* allows the receiver to accept out-of-sequence segments. Furthermore, when a *selective repeat* sender detects losses, it only retransmits the segments that have been lost and not the segments that have already been correctly received.

A *selective repeat* receiver maintains a sliding window of *W* segments and stores in a buffer the out-of-sequence segments that it receives. The figure below shows a five segment receive window on a receiver that has already received segments *7* and *9*.

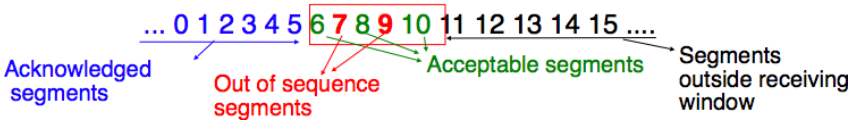

Figure 4.18: The receiving window with selective repeat

A *selective repeat* receiver discards all segments having an invalid CRC, and maintains the variable *lastack* as the sequence number of the last in-sequence segment that it has received. The receiver always includes the value of *lastack* in the acknowledgements that it sends. Some protocols also allow the *selective repeat* receiver to acknowledge the out-of-sequence segments that it has received. This can be done for example by placing the list of the sequence numbers of the correctly received, but out-of-sequence segments in the acknowledgements together with the *lastack* value.

When a *selective repeat* receiver receives a data segment, it first verifies whether the segment is inside its receiving window. If yes, the segment is placed in the receive buffer. If not, the received segment is discarded and an acknowledgement containing *lastack* is sent to the sender. The receiver then removes all consecutive segments starting at *lastack* (if any) from the receive buffer. The payloads of these segments are delivered to the user, *lastack* and the receiving window are updated, and an acknowledgement acknowledging the last segment received in sequence is sent.

The *selective repeat* sender maintains a sending buffer that can store up to *W* unacknowledged segments. These segments are sent as long as the sending buffer is not full. Several implementations of a *selective repeat* sender are possible. A simple implementation is to associate a retransmission timer to each segment. The timer is started when the segment is sent and cancelled upon reception of an acknowledgement that covers this segment. When a retransmission timer expires, the corresponding segment is retransmitted and this retransmission timer is restarted. When an acknowledgement is received, all the segments that are covered by this acknowledgement are removed from the sending buffer and the sliding window is updated.

The figure below illustrates the operation of *selective repeat* when segments are lost. In this figure, *C(OK,x)* is used to indicate that all segments, up to and including sequence number *x* have been received correctly.

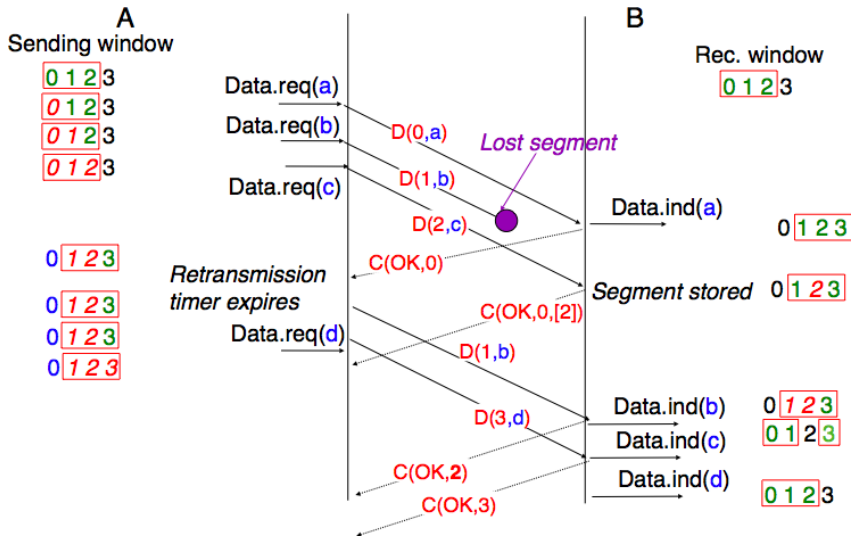

Figure 4.19: Selective repeat : example

Pure cumulative acknowledgements work well with the *go-back-n* strategy. However, with only cumulative acknowledgements a *selective repeat* sender cannot easily determine which data segments have been correctly received after a data segment has been lost. For example, in the figure above, the second *C(OK,0)* does not inform

explicitly the sender of the reception of *D(2,c)* and the sender could retransmit this segment although it has already been received. A possible solution to improve the performance of *selective repeat* is to provide additional information about the received segments in the acknowledgements that are returned by the receiver. For example, the receiver could add in the returned acknowledgement the list of the sequence numbers of all segments that have already been received. Such acknowledgements are sometimes called *selective acknowledgements*. This is illustrated in the figure below.

In the figure above, when the sender receives *C(OK,0,[2])*, it knows that all segments up to and including *D(0,...)* have been correctly received. It also knows that segment *D(2,...)* has been received and can cancel the retransmission timer associated to this segment. However, this segment should not be removed from the sending buffer before the reception of a cumulative acknowledgement (*C(OK,2)* in the figure above) that covers this segment.

Note: Maximum window size with *go-back-n* and *selective repeat*

A transport protocol that uses *n* bits to encode its sequence number can send up to 2^n different segments. However, to ensure a reliable delivery of the segments, *go-back-n* and *selective repeat* cannot use a sending window of 2^n segments. Consider first *go-back-n* and assume that a sender sends 2^n segments. These segments are received in-sequence by the destination, but all the returned acknowledgements are lost. The sender will retransmit all segments and they will all be accepted by the receiver and delivered a second time to the user. It is easy to see that this problem can be avoided if the maximum size of the sending window is $2^n - 1$ segments. A similar problem occurs with *selective repeat*. However, as the receiver accepts out-of-sequence segments, a sending window of $2^n - 1$ segments is not sufficient to ensure a reliable delivery of all segments. It can be easily shown that to avoid this problem, a *selective repeat* sender cannot use a window that is larger than $\frac{2^n}{2}$ segments.

Go-back-n or *selective repeat* are used by transport protocols to provide a reliable data transfer above an unreliable network layer service. Until now, we have assumed that the size of the sliding window was fixed for the entire lifetime of the connection. In practice a transport layer entity is usually implemented in the operating system and shares memory with other parts of the system. Furthermore, a transport layer entity must support several (possibly hundreds or thousands) of transport connections at the same time. This implies that the memory which can be used to support the sending or the receiving buffer of a transport connection may change during the lifetime of the connection [4] . Thus, a transport protocol must allow the sender and the receiver to adjust their window sizes.

To deal with this issue, transport protocols allow the receiver to advertise the current size of its receiving window in all the acknowledgements that it sends. The receiving window advertised by the receiver bounds the size of the sending buffer used by the sender. In practice, the sender maintains two state variables : *swin*, the size of its sending window (that may be adjusted by the system) and *rwin*, the size of the receiving window advertised by the receiver. At any time, the number of unacknowledged segments cannot be larger than *min(swin,rwin)* [5] . The utilisation of dynamic windows is illustrated in the figure below.

The receiver may adjust its advertised receive window based on its current memory consumption, but also to limit the bandwidth used by the sender. In practice, the receive buffer can also shrink as the application may not able to process the received data quickly enough. In this case, the receive buffer may be completely full and the advertised receive window may shrink to *0*. When the sender receives an acknowledgement with a receive window set to *0*, it is blocked until it receives an acknowledgement with a positive receive window. Unfortunately, as shown in the figure below, the loss of this acknowledgement could cause a deadlock as the sender waits for an acknowledgement while the receiver is waiting for a data segment.

To solve this problem, transport protocols rely on a special timer : the *persistence timer*. This timer is started by the sender whenever it receives an acknowledgement advertising a receive window set to *0*. When the timer expires, the sender retransmits an old segment in order to force the receiver to send a new acknowledgement, and hence send the current receive window size.

To conclude our description of the basic mechanisms found in transport protocols, we still need to discuss the impact of segments arriving in the wrong order. If two consecutive segments are reordered, the receiver relies on their sequence numbers to reorder them in its receive buffer. Unfortunately, as transport protocols reuse the same sequence number for different segments, if a segment is delayed for a prolonged period of time, it might still be accepted by the receiver. This is illustrated in the figure below where segment *D(1,b)* is delayed.

[4] For a discussion on how the sending buffer can change, see e.g. [SMM1998]

[5] Note that if the receive window shrinks, it might happen that the sender has already sent a segment that is not anymore inside its window. This segment will be discarded by the receiver and the sender will retransmit it later.

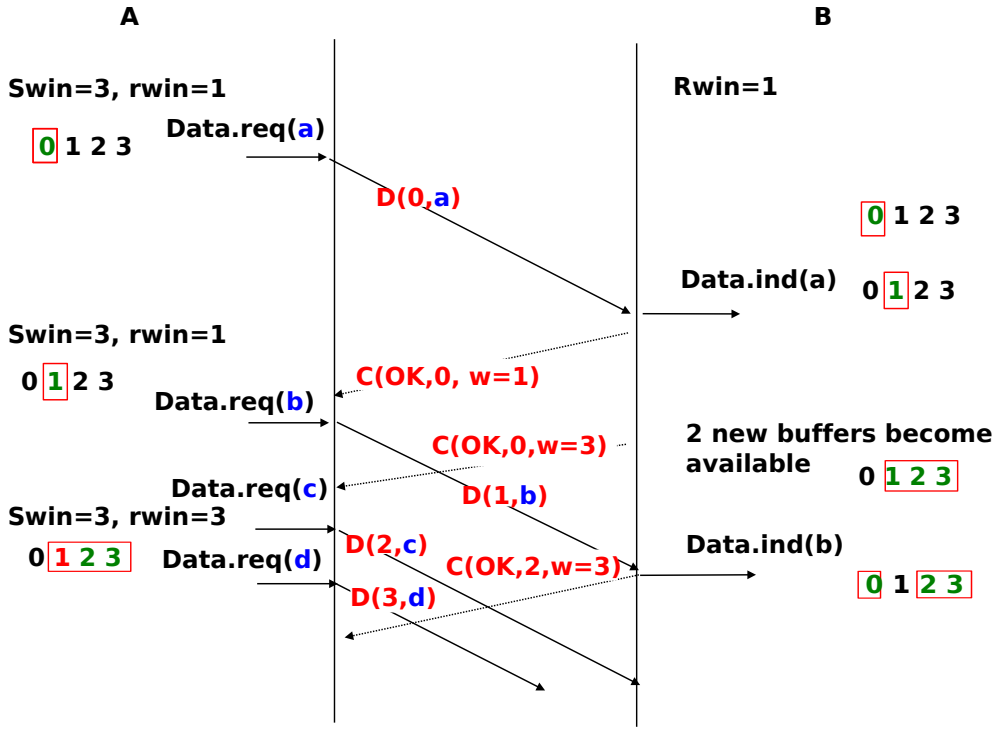

Figure 4.20: Dynamic receiving window

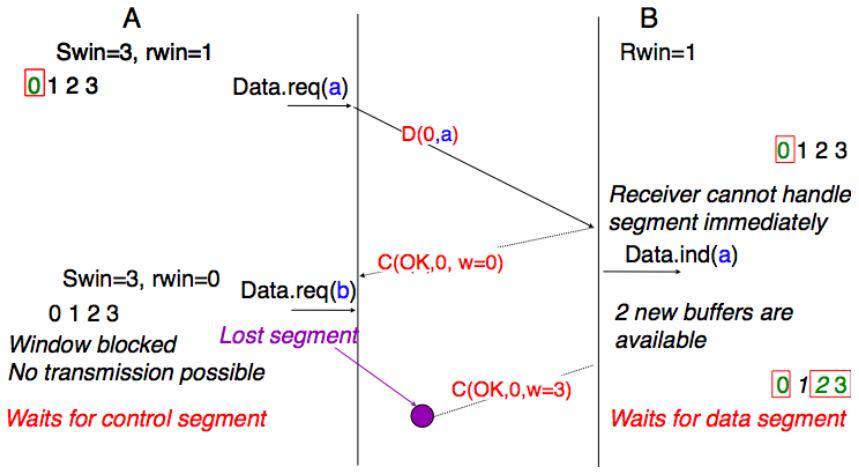

Figure 4.21: Risk of deadlock with dynamic windows

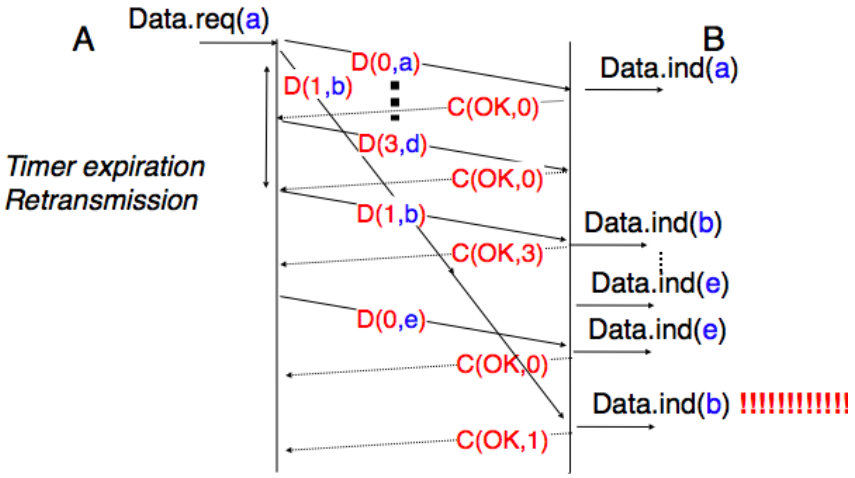

Figure 4.22: Ambiguities caused by excessive delays

To deal with this problem, transport protocols combine two solutions. First, they use 32 bits or more to encode the sequence number in the segment header. This increases the overhead, but also increases the delay between the transmission of two different segments having the same sequence number. Second, transport protocols require the network layer to enforce a *Maximum Segment Lifetime (MSL)*. The network layer must ensure that no packet remains in the network for more than MSL seconds. In the Internet the MSL is assumed [6] to be 2 minutes **RFC 793**. Note that this limits the maximum bandwidth of a transport protocol. If it uses n bits to encode its sequence numbers, then it cannot send more than 2^n segments every MSL seconds.

Transport protocols often need to send data in both directions. To reduce the overhead caused by the acknowledgements, most transport protocols use *piggybacking*. Thanks to this technique, a transport entity can place inside the header of the data segments that it sends, the acknowledgements and the receive window that it advertises for the opposite direction of the data flow. The main advantage of piggybacking is that it reduces the overhead as it is not necessary to send a complete segment to carry an acknowledgement. This is illustrated in the figure below where the acknowledgement number is underlined in the data segments. Piggybacking is only used when data flows in both directions. A receiver will generate a pure acknowledgement when it does not send data in the opposite direction as shown in the bottom of the figure.

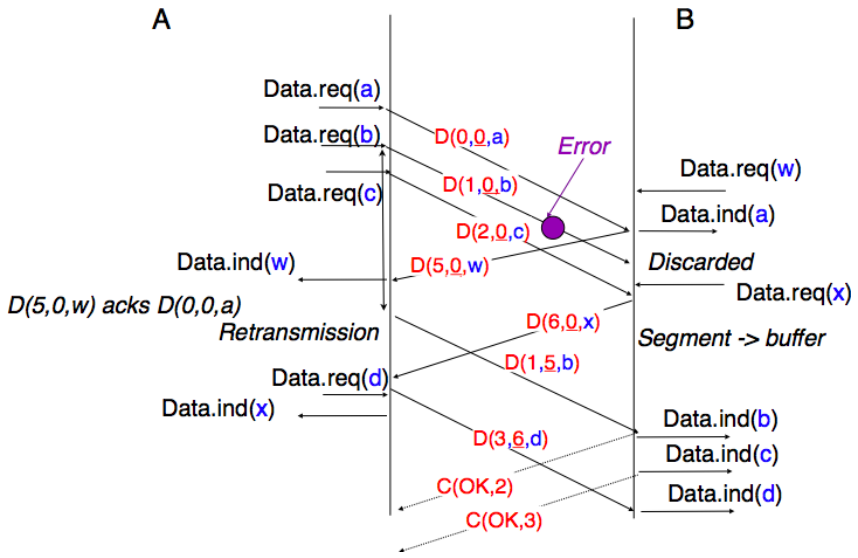

Figure 4.23: Piggybacking

[6] As we will see in the next chapter, the Internet does not strictly enforce this MSL. However, it is reasonable to expect that most packets on the Internet will not remain in the network during more than 2 minutes. There are a few exceptions to this rule, such as **RFC 1149** whose implementation is described in http://www.blug.linux.no/rfc1149/ but there are few real links supporting **RFC 1149** in the Internet.

The last point to be discussed about the data transfer mechanisms used by transport protocols is the provision of a byte stream service. As indicated in the first chapter, the byte stream service is widely used in the transport layer. The transport protocols that provide a byte stream service associate a sequence number to all the bytes that are sent and place the sequence number of the first byte of the segment in the segment's header. This is illustrated in the figure below. In this example, the sender chooses to put two bytes in each of the first three segments. This is due to graphical reasons, a real transport protocol would use larger segments in practice. However, the division of the byte stream into segments combined with the losses and retransmissions explain why the byte stream service does not preserve the SDU boundaries.

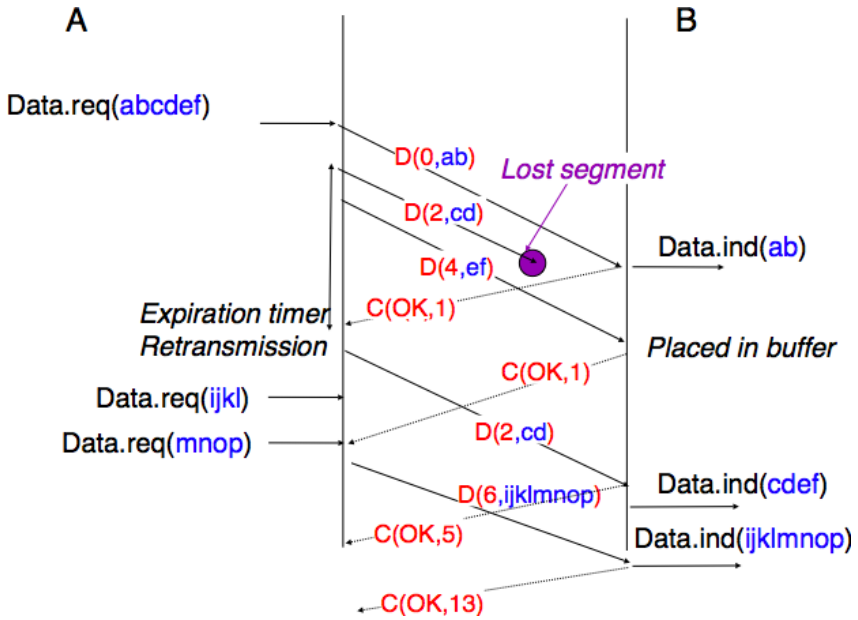

Figure 4.24: Provision of the byte stream service

Connection establishment and release

The last points to be discussed about the transport protocol are the mechanisms used to establish and release a transport connection.

We explained in the first chapters the service primitives used to establish a connection. The simplest approach to establish a transport connection would be to define two special control segments : *CR* and *CA*. The *CR* segment is sent by the transport entity that wishes to initiate a connection. If the remote entity wishes to accept the connection, it replies by sending a *CA* segment. The transport connection is considered to be established once the *CA* segment has been received and data segments can be sent in both directions.

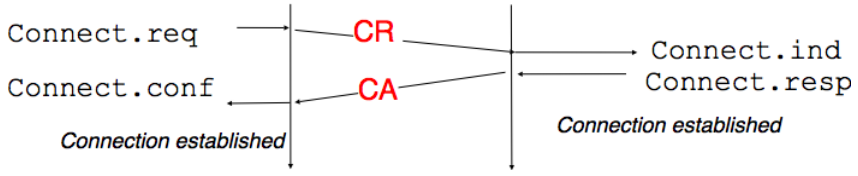

Figure 4.25: Naive transport connection establishment

Unfortunately, this scheme is not sufficient for several reasons. First, a transport entity usually needs to maintain several transport connections with remote entities. Sometimes, different users (i.e. processes) running above a given transport entity request the establishment of several transport connections to different users attached to the same remote transport entity. These different transport connections must be clearly separated to ensure that data from one connection is not passed to the other connections. This can be achieved by using a connection identifier, chosen by the transport entities and placed inside each segment to allow the entity which receives a segment to easily associate it to one established connection.

Second, as the network layer is imperfect, the *CR* or *CA* segment can be lost, delayed, or suffer from transmission errors. To deal with these problems, the control segments must be protected by using a CRC or checksum to detect transmission errors. Furthermore, since the *CA* segment acknowledges the reception of the *CR* segment, the *CR* segment can be protected by using a retransmission timer.

Unfortunately, this scheme is not sufficient to ensure the reliability of the transport service. Consider for example a short-lived transport connection where a single, but important transfer (e.g. money transfer from a bank account) is sent. Such a short-lived connection starts with a *CR* segment acknowledged by a *CA* segment, then the data segment is sent, acknowledged and the connection terminates. Unfortunately, as the network layer service is unreliable, delays combined to retransmissions may lead to the situation depicted in the figure below, where a delayed *CR* and data segments from a former connection are accepted by the receiving entity as valid segments, and the corresponding data is delivered to the user. Duplicating SDUs is not acceptable, and the transport protocol must solve this problem.

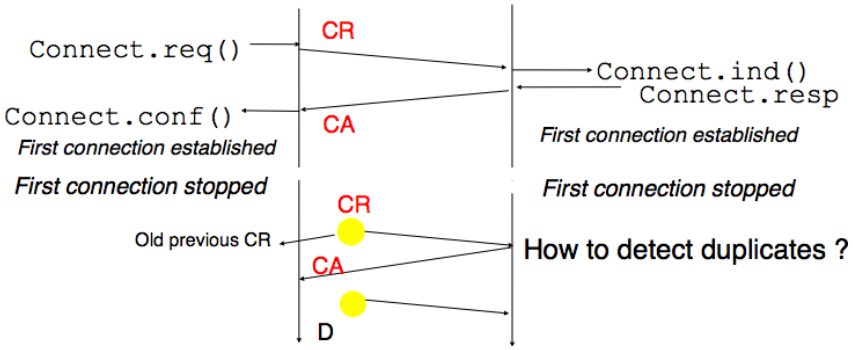

Figure 4.26: Duplicate transport connections ?

To avoid these duplicates, transport protocols require the network layer to bound the *Maximum Segment Lifetime (MSL)*. The organisation of the network must guarantee that no segment remains in the network for longer than *MSL* seconds. On today's Internet, *MSL* is expected to be 2 minutes. To avoid duplicate transport connections, transport protocol entities must be able to safely distinguish between a duplicate *CR* segment and a new *CR* segment, without forcing each transport entity to remember all the transport connections that it has established in the past.

A classical solution to avoid remembering the previous transport connections to detect duplicates is to use a clock inside each transport entity. This *transport clock* has the following characteristics :

- the *transport clock* is implemented as a k bits counter and its clock cycle is such that $2^k \times cycle >> MSL$. Furthermore, the *transport clock* counter is incremented every clock cycle and after each connection establishment. This clock is illustrated in the figure below.
- the *transport clock* must continue to be incremented even if the transport entity stops or reboots

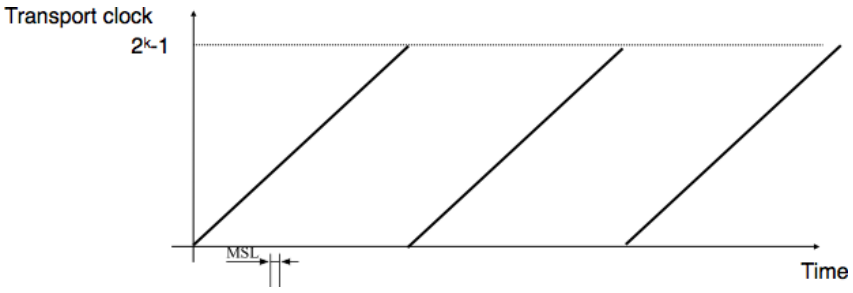

Figure 4.27: Transport clock

It should be noted that *transport clocks* do not need and usually are not synchronised to the real-time clock. Precisely synchronising real-time clocks is an interesting problem, but it is outside the scope of this document. See [Mills2006] for a detailed discussion on synchronising the real-time clock.

The *transport clock* is combined with an exchange of three segments, called the *three way handshake*, to detect duplicates. This *three way handshake* occurs as follows :

1. The initiating transport entity sends a *CR* segment. This segment requests the establishment of a transport connection. It contains a connection identifier (not shown in the figure) and a sequence number (*seq=x* in the figure below) whose value is extracted from the *transport clock* . The transmission of the *CR* segment is protected by a retransmission timer.
2. The remote transport entity processes the *CR* segment and creates state for the connection attempt. At this stage, the remote entity does not yet know whether this is a new connection attempt or a duplicate segment. It returns a *CA* segment that contains an acknowledgement number to confirm the reception of the *CR* segment (*ack=x* in the figure below) and a sequence number (*seq=y* in the figure below) whose value is extracted from its transport clock. At this stage, the connection is not yet established.
3. The initiating entity receives the *CA* segment. The acknowledgement number of this segment confirms that the remote entity has correctly received the *CA* segment. The transport connection is considered to be established by the initiating entity and the numbering of the data segments starts at sequence number *x*. Before sending data segments, the initiating entity must acknowledge the received *CA* segments by sending another *CA* segment.
4. The remote entity considers the transport connection to be established after having received the segment that acknowledges its *CA* segment. The numbering of the data segments sent by the remote entity starts at sequence number *y*.

The three way handshake is illustrated in the figure below.

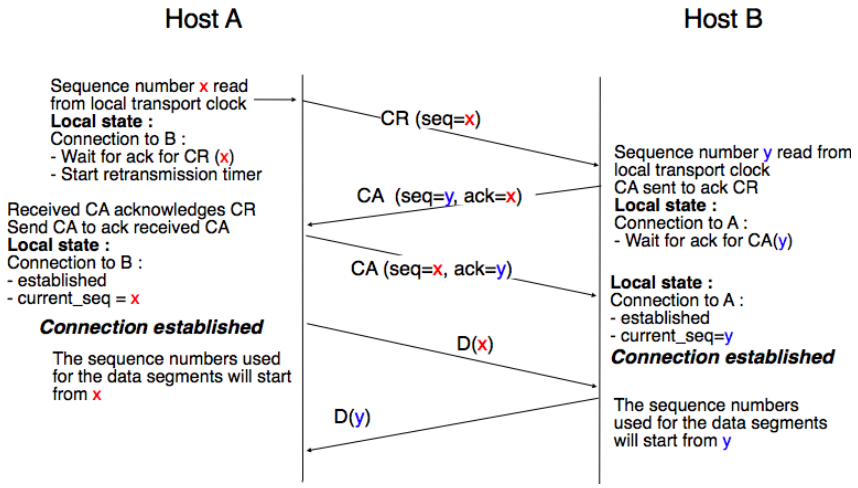

Figure 4.28: Three-way handshake

Thanks to the three way handshake, transport entities avoid duplicate transport connections. This is illustrated by the three scenarios below.

The first scenario is when the remote entity receives an old *CR* segment. It considers this *CR* segment as a connection establishment attempt and replies by sending a *CA* segment. However, the initiating host cannot match the received *CA* segment with a previous connection attempt. It sends a control segment (*REJECT* in the figure below) to cancel the spurious connection attempt. The remote entity cancels the connection attempt upon reception of this control segment.

A second scenario is when the initiating entity sends a *CR* segment that does not reach the remote entity and receives a duplicate *CA* segment from a previous connection attempt. This duplicate *CA* segment cannot contain a valid acknowledgement for the *CR* segment as the sequence number of the *CR* segment was extracted from the transport clock of the initiating entity. The *CA* segment is thus rejected and the *CR* segment is retransmitted upon expiration of the retransmission timer.

The last scenario is less likely, but it it important to consider it as well. The remote entity receives an old *CR* segment. It notes the connection attempt and acknowledges it by sending a *CA* segment. The initiating entity

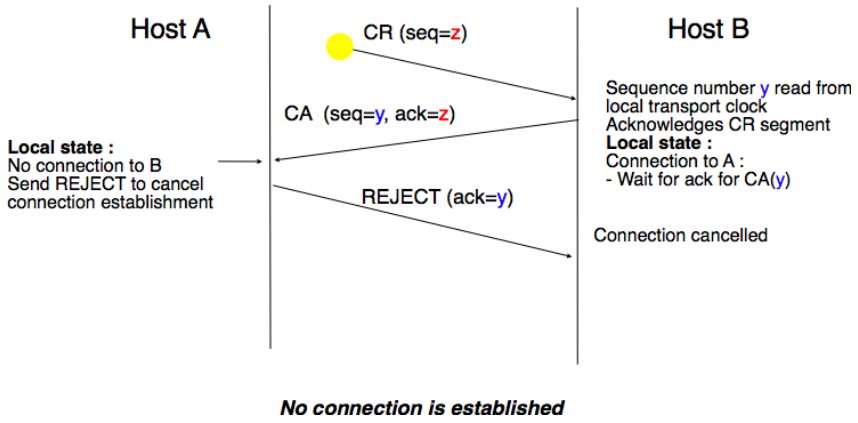

Figure 4.29: Three-way handshake : recovery from a duplicate *CR*

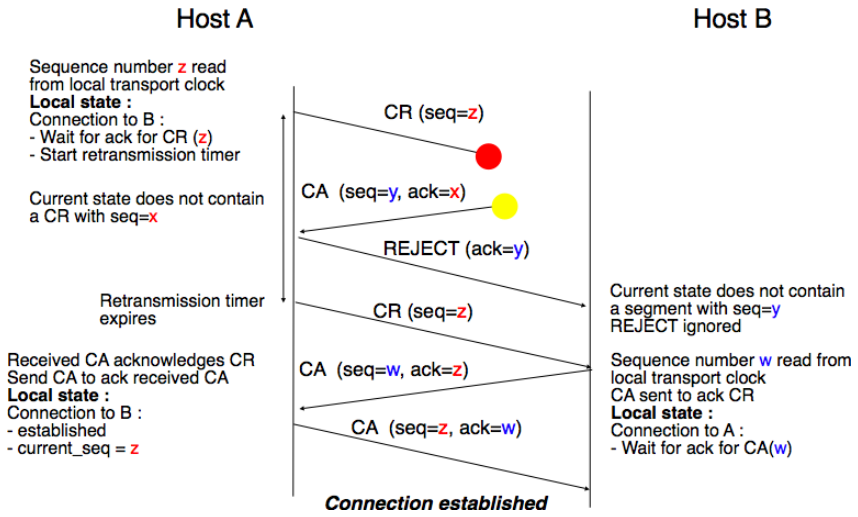

Figure 4.30: Three-way handshake : recovery from a duplicate *CA*

does not have a matching connection attempt and replies by sending a *REJECT*. Unfortunately, this segment never reaches the remote entity. Instead, the remote entity receives a retransmission of an older *CA* segment that contains the same sequence number as the first *CR* segment. This *CA* segment cannot be accepted by the remote entity as a confirmation of the transport connection as its acknowledgement number cannot have the same value as the sequence number of the first *CA* segment.

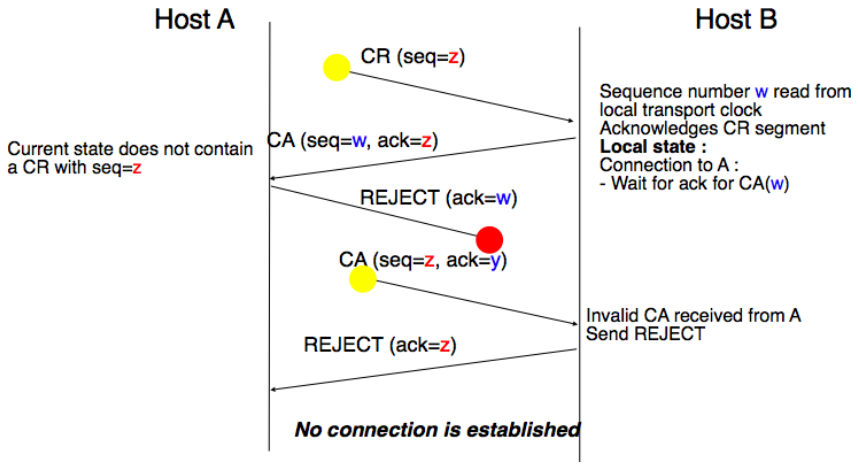

Figure 4.31: Three-way handshake : recovery from duplicates *CR* and *CA*

When we discussed the connection-oriented service, we mentioned that there are two types of connection releases : *abrupt release* and *graceful release*.

The first solution to release a transport connection is to define a new control segment (e.g. the *DR* segment) and consider the connection to be released once this segment has been sent or received. This is illustrated in the figure below.

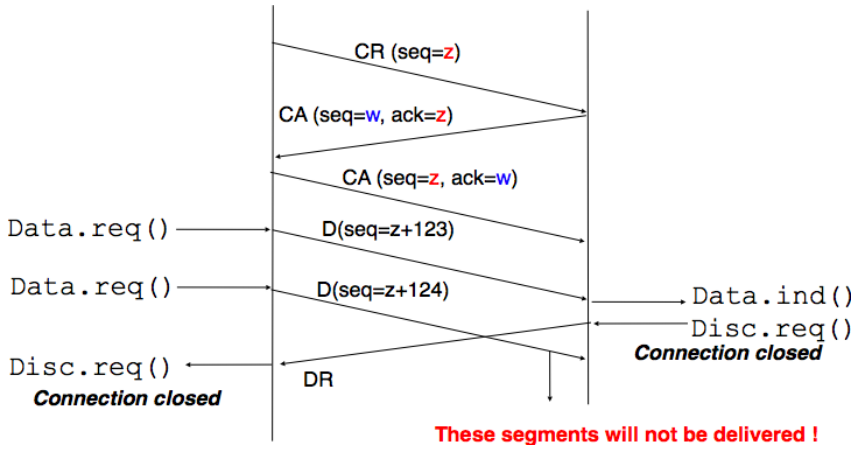

Figure 4.32: Abrupt connection release

As the entity that sends the *DR* segment cannot know whether the other entity has already sent all its data on the connection, SDUs can be lost during such an *abrupt connection release*.

The second method to release a transport connection is to release independently the two directions of data transfer. Once a user of the transport service has sent all its SDUs, it performs a *DISCONNECT.req* for its direction of data transfer. The transport entity sends a control segment to request the release of the connection *after* the delivery of all previous SDUs to the remote user. This is usually done by placing in the *DR* the next sequence number and by delivering the *DISCONNECT.ind* only after all previous *DATA.ind*. The remote entity confirms the reception of the *DR* segment and the release of the corresponding direction of data transfer by returning an acknowledgement. This is illustrated in the figure below.

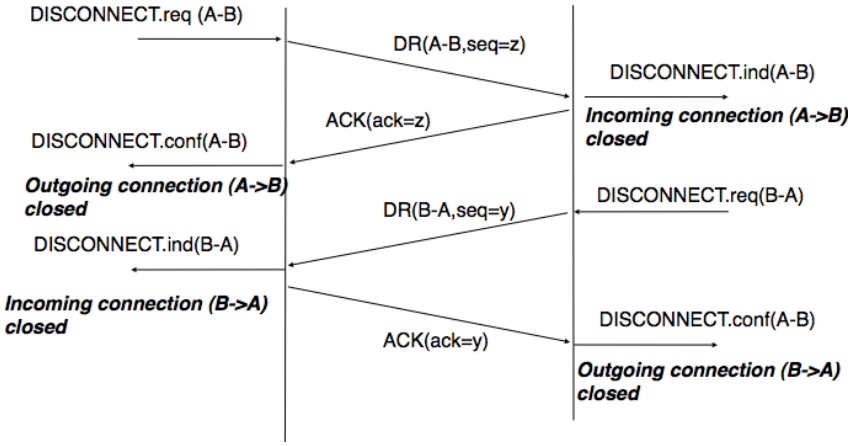

Figure 4.33: Graceful connection release

4.2 The User Datagram Protocol

The User Datagram Protocol (UDP) is defined in **RFC 768**. It provides an unreliable connectionless transport service on top of the unreliable network layer connectionless service. The main characteristics of the UDP service are :

- the UDP service cannot deliver SDUs that are larger than 65507 bytes [7]
- the UDP service does not guarantee the delivery of SDUs (losses and desquencing can occur)
- the UDP service will not deliver a corrupted SDU to the destination

Compared to the connectionless network layer service, the main advantage of the UDP service is that it allows several applications running on a host to exchange SDUs with several other applications running on remote hosts. Let us consider two hosts, e.g. a client and a server. The network layer service allows the client to send information to the server, but if an application running on the client wants to contact a particular application running on the server, then an additional addressing mechanism is required other than the IP address that identifies a host, in order to differentiate the application running on a host. This additional addressing is provided by *port numbers*. When a server using UDP is enabled on a host, this server registers a *port number*. This *port number* will be used by the clients to contact the server process via UDP.

The figure below shows a typical usage of the UDP port numbers. The client process uses port number *1234* while the server process uses port number *5678*. When the client sends a request, it is identified as originating from port number *1234* on the client host and destined to port number *5678* on the server host. When the server process replies to this request, the server's UDP implementation will send the reply as originating from port *5678* on the server host and destined to port *1234* on the client host.

UDP uses a single segment format shown in the figure below.

The UDP header contains four fields :

- a 16 bits source port
- a 16 bits destination port
- a 16 bits length field
- a 16 bits checksum

As the port numbers are encoded as a 16 bits field, there can be up to only 65535 different server processes that are bound to a different UDP port at the same time on a given server. In practice, this limit is never reached. However, it is worth noticing that most implementations divide the range of allowed UDP port numbers into three different ranges :

[7] This limitation is due to the fact that the network layer (IPv4 and IPv6) cannot transport packets that are larger than 64 KBytes. As UDP does not include any segmentation/reassembly mechanism, it cannot split a SDU before sending it.

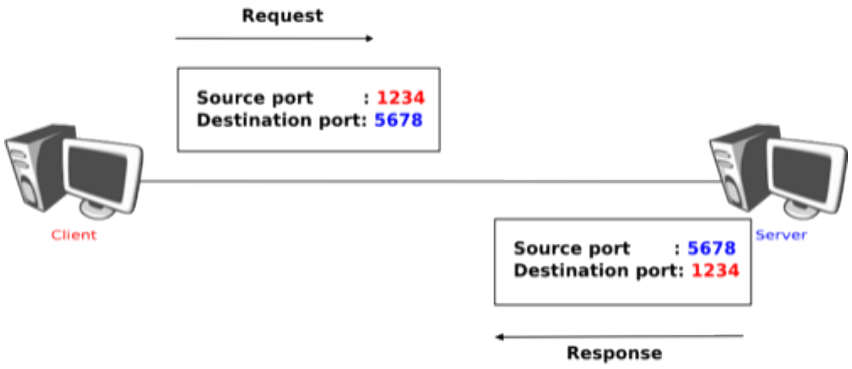

Figure 4.34: Usage of the UDP port numbers

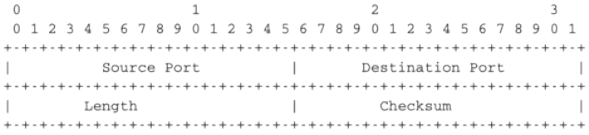

Figure 4.35: UDP Header Format

- the privileged port numbers (1 < port < 1024)
- the ephemeral port numbers (officially [8] 49152 <= port <= 65535)
- the registered port numbers (officially 1024 <= port < 49152)

In most Unix variants, only processes having system administrator privileges can be bound to port numbers smaller than *1024*. Well-known servers such as *DNS*, *NTP* or *RPC* use privileged port numbers. When a client needs to use UDP, it usually does not require a specific port number. In this case, the UDP implementation will allocate the first available port number in the ephemeral range. The range of registered port numbers should be used by servers. In theory, developers of network servers should register their port number officially through IANA, but few developers do this.

Note: Computation of the UDP checksum

The checksum of the UDP segment is computed over :

- a pseudo header containing the source IP address, the destination IP address and a 32 bits bit field containing the most significant byte set to 0, the second set to 17 and the length of the UDP segment in the lower two bytes
- the entire UDP segment, including its header

This pseudo-header allows the receiver to detect errors affecting the IP source or destination addresses placed in the IP layer below. This is a violation of the layering principle that dates from the time when UDP and IP were elements of a single protocol. It should be noted that if the checksum algorithm computes value '0x0000', then value '0xffff' is transmitted. A UDP segment whose checksum is set to '0x0000' is a segment for which the transmitter did not compute a checksum upon transmission. Some *NFS* servers chose to disable UDP checksums for performance reasons, but this caused problems that were difficult to diagnose. In practice, there are rarely good reasons to disable UDP checksums. A detailed discussion of the implementation of the Internet checksum may be found in **RFC 1071**

Several types of applications rely on UDP. As a rule of thumb, UDP is used for applications where delay must be minimised or losses can be recovered by the application itself. A first class of the UDP-based applications are applications where the client sends a short request and expects a quick and short answer. The *DNS* is an example of

[8] A discussion of the ephemeral port ranges used by different TCP/UDP implementations may be found in http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html

a UDP application that is often used in the wide area. However, in local area networks, many distributed systems rely on Remote Procedure Call (*RPC*) that is often used on top of UDP. In Unix environments, the Network File System (*NFS*) is built on top of RPC and runs frequently on top of UDP. A second class of UDP-based applications are the interactive computer games that need to frequently exchange small messages, such as the player's location or their recent actions. Many of these games use UDP to minimise the delay and can recover from losses. A third class of applications are multimedia applications such as interactive Voice over IP or interactive Video over IP. These interactive applications expect a delay shorter than about 200 milliseconds between the sender and the receiver and can recover from losses directly inside the application.

4.3 The Transmission Control Protocol

The Transmission Control Protocol (TCP) was initially defined in **RFC 793**. Several parts of the protocol have been improved since the publication of the original protocol specification [9]. However, the basics of the protocol remain and an implementation that only supports **RFC 793** should inter-operate with today's implementation.

TCP provides a reliable bytestream, connection-oriented transport service on top of the unreliable connectionless network service provided by *IP*. TCP is used by a large number of applications, including :

- Email (*SMTP*, *POP*, *IMAP*)
- World wide web (*HTTP*, ...)
- Most file transfer protocols (*ftp*, peer-to-peer file sharing applications , ...)
- remote computer access : *telnet*, *ssh*, *X11*, *VNC*, ...
- non-interactive multimedia applications : flash

On the global Internet, most of the applications used in the wide area rely on TCP. Many studies [10] have reported that TCP was responsible for more than 90% of the data exchanged in the global Internet.

To provide this service, TCP relies on a simple segment format that is shown in the figure below. Each TCP segment contains a header described below and, optionally, a payload. The default length of the TCP header is twenty bytes, but some TCP headers contain options.

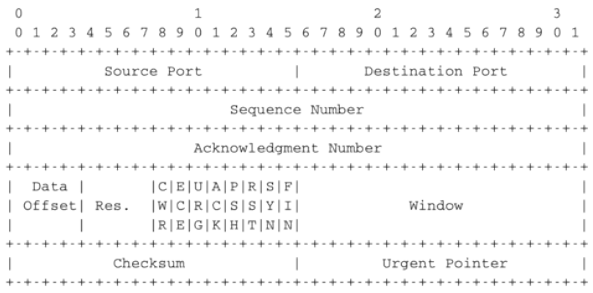

Figure 4.36: TCP header format

A TCP header contains the following fields :

- Source and destination ports. The source and destination ports play an important role in TCP, as they allow the identification of the connection to which a TCP segment belongs. When a client opens a TCP connection, it typically selects an ephemeral TCP port number as its source port and contacts the server by using the server's port number. All the segments that are sent by the client on this connection have the same source and destination ports. The server sends segments that contain as source (resp. destination port, the

[9] A detailed presentation of all standardisation documents concerning TCP may be found in **RFC 4614**

[10] Several researchers have analysed the utilisation of TCP and UDP in the global Internet. Most of these studies have been performed by collecting all the packets transmitted over a given link during a period of a few hours or days and then analysing their headers to infer the transport protocol used, the type of application, ... Recent studies include http://www.caida.org/research/traffic-analysis/tcpudpratio/, https://research.sprintlabs.com/packstat/packetoverview.php or http://www.nanog.org/meetings/nanog43/presentations/Labovitz_internetstats_N43.pdf

destination (resp. source) port of the segments sent by the client (see figure *Utilization of the TCP source and destination ports*). A TCP connection is always identified by five pieces of information :

  - the IP address of the client
  - the IP address of the server
  - the port chosen by the client
  - the port chosen by the server
  - TCP

- the *sequence number* (32 bits), *acknowledgement number* (32 bits) and *window* (16 bits) fields are used to provide a reliable data transfer, using a window-based protocol. In a TCP bytestream, each byte of the stream consumes one sequence number. Their utilisation will be described in more detail in section *TCP reliable data transfer*
- the *Urgent pointer* is used to indicate that some data should be considered as urgent in a TCP bytestream. However, it is rarely used in practice and will not be described here. Additional details about the utilisation of this pointer may be found in **RFC 793**, **RFC 1122** or [Stevens1994]
- the flags field contains a set of bit flags that indicate how a segment should be interpreted by the TCP entity receiving it :
  - the *SYN* flag is used during connection establishment
  - the *FIN* flag is used during connection release
  - the *RST* is used in case of problems or when an invalid segment has been received
  - when the *ACK* flag is set, it indicates that the *acknowledgment* field contains a valid number. Otherwise, the content of the *acknowledgment* field must be ignored by the receiver
  - the *URG* flag is used together with the *Urgent pointer*
  - the *PSH* flag is used as a notification from the sender to indicate to the receiver that it should pass all the data it has received to the receiving process. However, in practice TCP implementations do not allow TCP users to indicate when the *PSH* flag should be set and thus there are few real utilizations of this flag.
- the *checksum* field contains the value of the Internet checksum computed over the entire TCP segment and a pseudo-header as with UDP
- the *Reserved* field was initially reserved for future utilization. It is now used by **RFC 3168**.
- the *TCP Header Length* (THL) or *Data Offset* field is a four bits field that indicates the size of the TCP header in 32 bit words. The maximum size of the TCP header is thus 64 bytes.
- the *Optional header extension* is used to add optional information to the TCP header. Thanks to this header extension, it is possible to add new fields to the TCP header that were not planned in the original specification. This allowed TCP to evolve since the early eighties. The details of the TCP header extension are explained in sections *TCP connection establishment* and *TCP reliable data transfer*.

The rest of this section is organised as follows. We first explain the establishment and the release of a TCP connection, then we discuss the mechanisms that are used by TCP to provide a reliable bytestream service. We end the section with a discussion of network congestion and explain the mechanisms that TCP uses to avoid congestion collapse.

4.3.1 TCP connection establishment

A TCP connection is established by using a three-way handshake. The connection establishment phase uses the *sequence number*, the *acknowledgment number* and the *SYN* flag. When a TCP connection is established, the two communicating hosts negotiate the initial sequence number to be used in both directions of the connection. For this, each TCP entity maintains a 32 bits counter, which is supposed to be incremented by one at least every 4

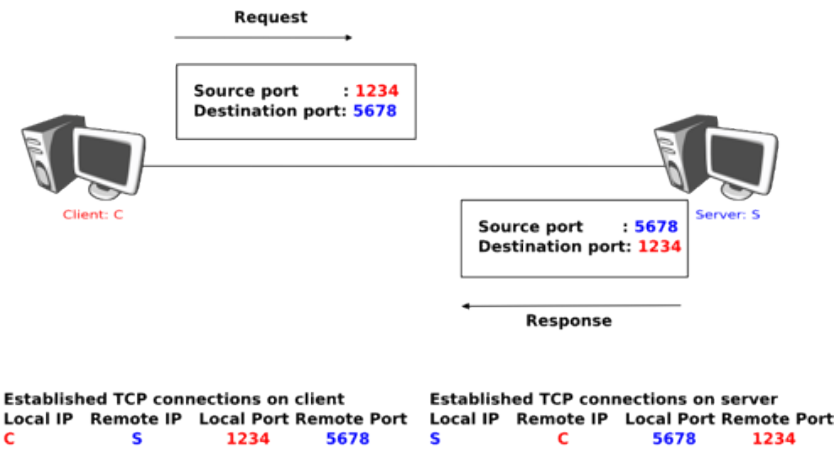

Figure 4.37: Utilization of the TCP source and destination ports

microseconds and after each connection establishment [11]. When a client host wants to open a TCP connection with a server host, it creates a TCP segment with :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the client host's TCP entity

Upon reception of this segment (which is often called a *SYN segment*), the server host replies with a segment containing :

- the *SYN* flag set
- the *sequence number* set to the current value of the 32 bits counter of the server host's TCP entity
- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN* segment incremented by 1 ($mod\ 2^{32}$). When a TCP entity sends a segment having *x+1* as acknowledgment number, this indicates that it has received all data up to and including sequence number *x* and that it is expecting data having sequence number *x+1*. As the *SYN* flag was set in a segment having sequence number *x*, this implies that setting the *SYN* flag in a segment consumes one sequence number.

This segment is often called a *SYN+ACK* segment. The acknowledgment confirms to the client that the server has correctly received the *SYN* segment. The *sequence number* of the *SYN+ACK* segment is used by the server host to verify that the *client* has received the segment. Upon reception of the *SYN+ACK* segment, the client host replies with a segment containing :

- the *ACK* flag set
- the *acknowledgment number* set to the *sequence number* of the received *SYN+ACK* segment incremented by 1 ($mod\ 2^{32}$)

At this point, the TCP connection is open and both the client and the server are allowed to send TCP segments containing data. This is illustrated in the figure below.

In the figure above, the connection is considered to be established by the client once it has received the *SYN+ACK* segment, while the server considers the connection to be established upon reception of the *ACK* segment. The first data segment sent by the client (server) has its *sequence number* set to *x+1* (resp. *y+1*).

Note: Computing TCP's initial sequence number

In the original TCP specification **RFC 793**, each TCP entity maintained a clock to compute the initial sequence number (*ISN*) placed in the *SYN* and *SYN+ACK* segments. This made the ISN predictable and caused a security issue. The typical security problem was the following. Consider a server that trusts a host based on its IP address

[11] This 32 bits counter was specified in **RFC 793**. A 32 bits counter that is incremented every 4 microseconds wraps in about 4.5 hours. This period is much larger than the Maximum Segment Lifetime that is fixed at 2 minutes in the Internet (**RFC 791**, **RFC 1122**).

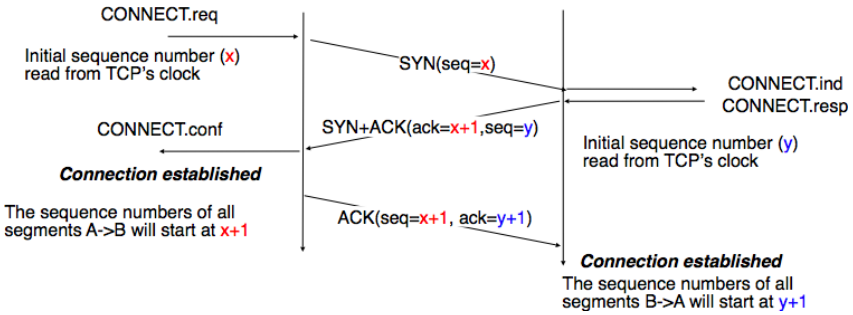

Figure 4.38: Establishment of a TCP connection

and allows the system administrator to login from this host without giving a password [12]. Consider now an attacker who knows this particular configuration and is able to send IP packets having the client's address as source. He can send fake TCP segments to the server, but does not receive the server's answers. If he can predict the *ISN* that is chosen by the server, he can send a fake *SYN* segment and shortly after the fake *ACK* segment confirming the reception of the *SYN+ACK* segment sent by the server. Once the TCP connection is open, he can use it to send any command to the server. To counter this attack, current TCP implementations add randomness to the *ISN*. One of the solutions, proposed in **RFC 1948** is to compute the *ISN* as

```
ISN = M + H(localhost, localport, remotehost, remoteport, secret).
```

where *M* is the current value of the TCP clock and *H'is a cryptographic hash function. 'localhost* and *remotehost* (resp. *localport* and *remoteport*) are the IP addresses (port numbers) of the local and remote host and *secret* is a random number only known by the server. This method allows the server to use different ISNs for different clients at the same time. Measurements performed with the first implementations of this technique showed that it was difficult to implement it correctly, but today's TCP implementation now generate good ISNs.

A server could, of course, refuse to open a TCP connection upon reception of a *SYN* segment. This refusal may be due to various reasons. There may be no server process that is listening on the destination port of the *SYN* segment. The server could always refuse connection establishments from this particular client (e.g. due to security reasons) or the server may not have enough resources to accept a new TCP connection at that time. In this case, the server would reply with a TCP segment having its *RST* flag set and containing the *sequence number* of the received *SYN* segment as its *acknowledgment number*. This is illustrated in the figure below. We discuss the other utilizations of the TCP *RST* flag later (see *TCP connection release*).

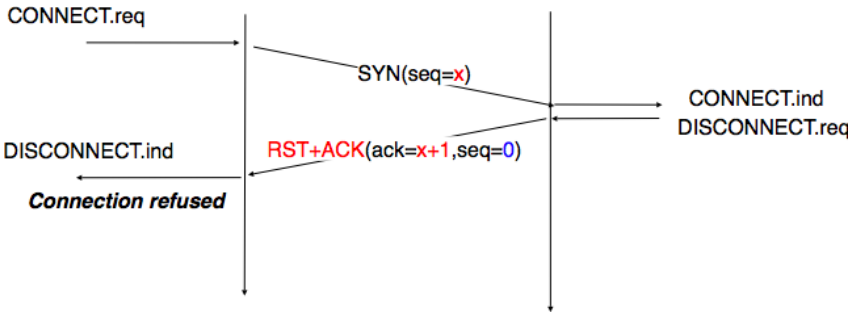

Figure 4.39: TCP connection establishment rejected by peer

TCP connection establishment can be described as the four state Finite State Machine shown below. In this FSM, *!X* (resp. *?Y*) indicates the transmission of segment *X* (resp. reception of segment *Y*) during the corresponding transition. *Init* is the initial state.

A client host starts in the *Init* state. It then sends a *SYN* segment and enters the *SYN Sent* state where it waits for a *SYN+ACK* segment. Then, it replies with an *ACK* segment and enters the *Established* state where data can

[12] On many departmental networks containing Unix workstations, it was common to allow users on one of the hosts to use rlogin **RFC 1258** to run commands on any of the workstations of the network without giving any password. In this case, the remote workstation "authenticated" the client host based on its IP address. This was a bad practice from a security viewpoint.

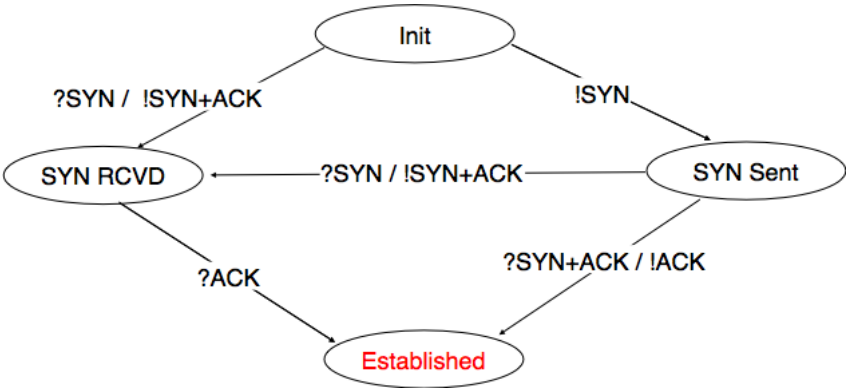

Figure 4.40: TCP FSM for connection establishment

be exchanged. On the other hand, a server host starts in the *Init* state. When a server process starts to listen to a destination port, the underlying TCP entity creates a TCP control block and a queue to process incoming *SYN* segments. Upon reception of a *SYN* segment, the server's TCP entity replies with a *SYN+ACK* and enters the *SYN RCVD* state. It remains in this state until it receives an *ACK* segment that acknowledges its *SYN+ACK* segment, with this it then enters the Established state.

Apart from these two paths in the TCP connection establishment FSM, there is a third path that corresponds to the case when both the client and the server send a *SYN* segment to open a TCP connection [13]. In this case, the client and the server send a *SYN* segment and enter the *SYN Sent* state. Upon reception of the *SYN* segment sent by the other host, they reply by sending a *SYN+ACK* segment and enter the *SYN RCVD* state. The *SYN+ACK* that arrives from the other host allows it to transition to the *Established* state. The figure below illustrates such a simultaneous establishment of a TCP connection.

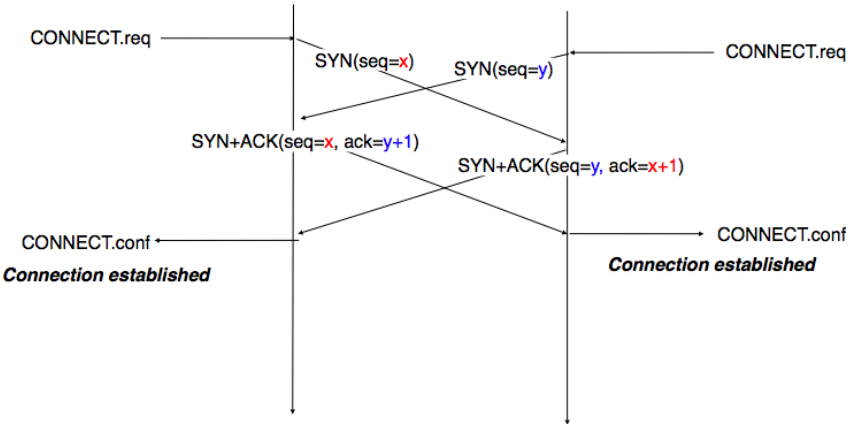

Figure 4.41: Simultaneous establishment of a TCP connection

[13] Of course, such a simultaneous TCP establishment can only occur if the source port chosen by the client is equal to the destination port chosen by the server. This may happen when a host can serve both as a client as a server or in peer-to-peer applications when the communicating hosts do not use ephemeral port numbers.

Denial of Service attacks

When a TCP entity opens a TCP connection, it creates a Transmission Control Block (*TCB*). The TCB contains the entire state that is maintained by the TCP entity for each TCP connection. During connection establishment, the TCB contains the local IP address, the remote IP address, the local port number, the remote port number, the current local sequence number, the last sequence number received from the remote entity. Until the mid 1990s, TCP implementations had a limit on the number of TCP connections that could be in the *SYN RCVD* state at a given time. Many implementations set this limit to about 100 TCBs. This limit was considered sufficient even for heavily load http servers given the small delay between the reception of a *SYN* segment and the reception of the *ACK* segment that terminates the establishment of the TCP connection. When the limit of 100 TCBs in the *SYN Rcvd* state is reached, the TCP entity discards all received TCP *SYN* segments that do not correspond to an existing TCB.
This limit of 100 TCBs in the *SYN Rcvd* state was chosen to protect the TCP entity from the risk of overloading its memory with too many TCBs in the *SYN Rcvd* state. However, it was also the reason for a new type of Denial of Service (DoS) attack **RFC 4987**. A DoS attack is defined as an attack where an attacker can render a resource unavailable in the network. For example, an attacker may cause a DoS attack on a 2 Mbps link used by a company by sending more than 2 Mbps of packets through this link. In this case, the DoS attack was more subtle. As a TCP entity discards all received *SYN* segments as soon as it has 100 TCBs in the *SYN Rcvd* state, an attacker simply had to send a few 100 *SYN* segments every second to a server and never reply to the received *SYN+ACK* segments. To avoid being caught, attackers were of course sending these *SYN* segments with a different address than their own IP address [a]. On most TCP implementations, once a TCB entered the *SYN Rcvd* state, it remained in this state for several seconds, waiting for a retransmission of the initial *SYN* segment. This attack was later called a *SYN flood* attack and the servers of the ISP named panix were among the first to be affected by this attack.
To avoid the *SYN flood* attacks, recent TCP implementations no longer enter the *SYN Rcvd* state upon reception of a *SYN segment*. Instead, they reply directly with a *SYN+ACK* segment and wait until the reception of a valid *ACK*. This implementation trick is only possible if the TCP implementation is able to verify that the received *ACK* segment acknowledges the *SYN+ACK* segment sent earlier without storing the initial sequence number of this *SYN+ACK* segment in a TCB. The solution to solve this problem, which is known as SYN cookies is to compute the 32 bits of the *ISN* as follows :

- the high order bits contain the low order bits of a counter that is incremented slowly
- the low order bits contain a hash value computed over the local and remote IP addresses and ports and a random secret only known to the server

The advantage of the SYN cookies is that by using them, the server does not need to create a *TCB* upon reception of the *SYN* segment and can still check the returned *ACK* segment by recomputing the *SYN cookie*.

[a] Sending a packet with a different source IP address than the address allocated to the host is called sending a *spoofed packet*.

Retransmitting the first *SYN* segment

As IP provides an unreliable connectionless service, the *SYN* and *SYN+ACK* segments sent to open a TCP connection could be lost. Current TCP implementations start a retransmission timer when they send the first *SYN* segment. This timer is often set to three seconds for the first retransmission and then doubles after each retransmission **RFC 2988**. TCP implementations also enforce a maximum number of retransmissions for the initial *SYN* segment.

As explained earlier, TCP segments may contain an optional header extension. In the *SYN* and *SYN+ACK* segments, these options are used to negotiate some parameters and the utilisation of extensions to the basic TCP specification.

The first parameter which is negotiated during the establishment of a TCP connection is the Maximum Segment Size (*MSS*). The MSS is the size of the largest segment that a TCP entity is able to process. According to **RFC 879**, all TCP implementations must be able to receive TCP segments containing 536 bytes of payload. However, most TCP implementations are able to process larger segments. Such TCP implementations use the TCP MSS Option in the *SYN*/*SYN+ACK* segment to indicate the largest segment they are able to process. The MSS value indicates the maximum size of the payload of the TCP segments. The client (resp. server) stores in its *TCB* the MSS value announced by the server (resp. the client).

Another utilisation of TCP options during connection establishment is to enable TCP extensions. For example, consider **RFC 1323** (which is discussed in *TCP reliable data transfer*). **RFC 1323** defines TCP extensions to support timestamps and larger windows. If the client supports **RFC 1323**, it adds a **RFC 1323** option to its *SYN* segment. If the server understands this **RFC 1323** option and wishes to use it, it replies with an **RFC 1323** option in the *SYN+ACK* segment and the extension defined in **RFC 1323** is used throughout the TCP connection. Otherwise, if the server's *SYN+ACK* does not contain the **RFC 1323** option, the client is not allowed to use this extension and the corresponding TCP header options throughout the TCP connection. TCP's option mechanism is flexible and it allows the extension of TCP while maintaining compatibility with older implementations.

The TCP options are encoded by using a Type Length Value format where :

- the first byte indicates the *type* of the option.
- the second byte indicates the total length of the option (including the first two bytes) in bytes
- the last bytes are specific for each type of option

RFC 793 defines the Maximum Segment Size (MSS) TCP option that must be understood by all TCP implementations. This option (type 2) has a length of 4 bytes and contains a 16 bits word that indicates the MSS supported by the sender of the *SYN* segment. The MSS option can only be used in TCP segments having the *SYN* flag set.

RFC 793 also defines two special options that must be supported by all TCP implementations. The first option is *End of option*. It is encoded as a single byte having value *0x00* and can be used to ensure that the TCP header extension ends on a 32 bits boundary. The *No-Operation* option, encoded as a single byte having value *0x01*, can be used when the TCP header extension contains several TCP options that should be aligned on 32 bit boundaries. All other options [14] are encoded by using the TLV format.

Note: The robustness principle

The handling of the TCP options by TCP implementations is one of the many applications of the *robustness principle* which is usually attributed to Jon Postel and is often quoted as *"Be liberal in what you accept, and conservative in what you send"* **RFC 1122**

Concerning the TCP options, the robustness principle implies that a TCP implementation should be able to accept TCP options that it does not understand, in particular in received *SYN* segments, and that it should be able to parse any received segment without crashing, even if the segment contains an unknown TCP option. Furthermore, a server should not send in the *SYN+ACK* segment or later, options that have not been proposed by the client in the *SYN* segment.

4.3.2 TCP connection release

TCP, like most connection-oriented transport protocols, supports two types of connection release :

- graceful connection release, where each TCP user can release its own direction of data transfer
- abrupt connection release, where either one user closes both directions of data transfer or one TCP entity is forced to close the connection (e.g. because the remote host does not reply anymore or due to lack of resources)

The abrupt connection release mechanism is very simple and relies on a single segment having the *RST* bit set. A TCP segment containing the *RST* bit can be sent for the following reasons :

- a non-*SYN* segment was received for a non-existing TCP connection **RFC 793**
- by extension, some implementations respond with an *RST* segment to a segment that is received on an existing connection but with an invalid header **RFC 3360**. This causes the corresponding connection to be closed and has caused security attacks **RFC 4953**
- by extension, some implementations send an *RST* segment when they need to close an existing TCP connection (e.g. because there are not enough resources to support this connection or because the remote host is considered to be unreachable). Measurements have shown that this usage of TCP *RST* was widespread [AW05]

[14] The full list of all TCP options may be found at http://www.iana.org/assignments/tcp-parameters/

When an *RST* segment is sent by a TCP entity, it should contain the current value of the *sequence number* for the connection (or 0 if it does not belong to any existing connection) and the *acknowledgement number* should be set to the next expected in-sequence *sequence number* on this connection.

Note: TCP *RST* wars

TCP implementers should ensure that two TCP entities never enter a TCP *RST* war where host *A* is sending a *RST* segment in response to a previous *RST* segment that was sent by host *B* in response to a TCP *RST* segment sent by host *A* ... To avoid such an infinite exchange of *RST* segments that do not carry data, a TCP entity is *never* allowed to send a *RST* segment in response to another *RST* segment.

The normal way of terminating a TCP connection is by using the graceful TCP connection release. This mechanism uses the *FIN* flag of the TCP header and allows each host to release its own direction of data transfer. As for the *SYN* flag, the utilisation of the *FIN* flag in the TCP header consumes one sequence number. The figure *FSM for TCP connection release* shows the part of the TCP FSM used when a TCP connection is released.

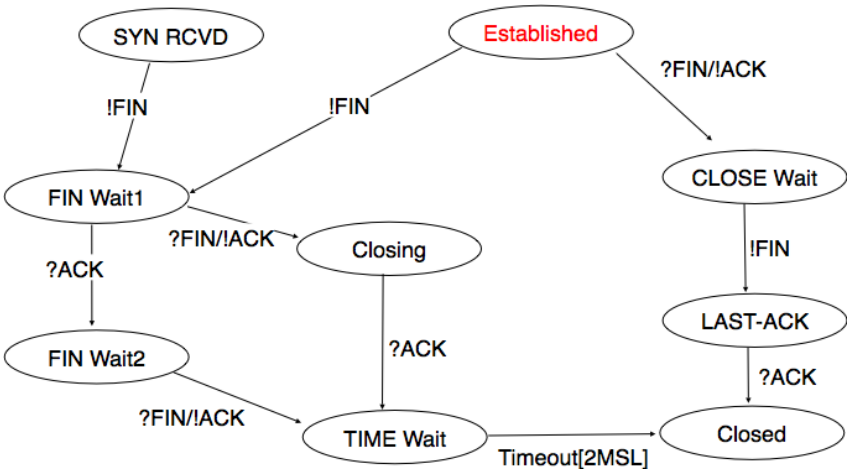

Figure 4.42: FSM for TCP connection release

Starting from the *Established* state, there are two main paths through this FSM.

The first path is when the host receives a segment with sequence number *x* and the *FIN* flag set. The utilisation of the *FIN* flag indicates that the byte before *sequence number x* was the last byte of the byte stream sent by the remote host. Once all of the data has been delivered to the user, the TCP entity sends an *ACK* segment whose *ack* field is set to $(x + 1)\ mod\ 2^{32}$ to acknowledge the *FIN* segment. The *FIN* segment is subject to the same retransmission mechanisms as a normal TCP segment. In particular, its transmission is protected by the retransmission timer. At this point, the TCP connection enters the *CLOSE_WAIT* state. In this state, the host can still send data to the remote host. Once all its data have been sent, it sends a *FIN* segment and enter the *LAST_ACK* state. In this state, the TCP entity waits for the acknowledgement of its *FIN* segment. It may still retransmit unacknowledged data segments e.g. if the retransmission timer expires. Upon reception of the acknowledgement for the *FIN* segment, the TCP connection is completely closed and its *TCB* can be discarded.

The second path is when the host decides first to send a *FIN* segment. In this case, it enters the *FIN_WAIT1* state. It this state, it can retransmit unacknowledged segments but cannot send new data segments. It waits for an acknowledgement of its *FIN* segment, but may receive a *FIN* segment sent by the remote host. In the first case, the TCP connection enters the *FIN_WAIT2* state. In this state, new data segments from the remote host are still accepted until the reception of the *FIN* segment. The acknowledgement for this *FIN* segment is sent once all data received before the *FIN* segment have been delivered to the user and the connection enters the *TIME_WAIT* state. In the second case, a *FIN* segment is received and the connection enters the *Closing* state once all data received from the remote host have been delivered to the user. In this state, no new data segments can be sent and the host waits for an acknowledgement of its *FIN* segment before entering the *TIME_WAIT* state.

The *TIME_WAIT* state is different from the other states of the TCP FSM. A TCP entity enters this state after having sent the last *ACK* segment on a TCP connection. This segment indicates to the remote host that all the data that it has sent have been correctly received and that it can safely release the TCP connection and discard

the corresponding *TCB*. After having sent the last *ACK* segment, a TCP connection enters the *TIME_WAIT* and remains in this state for $2 * MSL$ seconds. During this period, the TCB of the connection is maintained. This ensures that the TCP entity that sent the last *ACK* maintains enough state to be able to retransmit this segment if this *ACK* segment is lost and the remote host retransmits its last *FIN* segment or another one. The delay of $2 * MSL$ seconds ensures that any duplicate segments on the connection would be handled correctly without causing the transmission of an *RST* segment. Without the *TIME_WAIT* state and the $2 * MSL$ seconds delay, the connection release would not be graceful when the last *ACK* segment is lost.

Note: TIME_WAIT on busy TCP servers

The $2 * MSL$ seconds delay in the *TIME_WAIT* state is an important operational problem on servers having thousands of simultaneously opened TCP connections [FTY99]. Consider for example a busy web server that processes 10.000 TCP connections every second. If each of these connections remain in the *TIME_WAIT* state for 4 minutes, this implies that the server would have to maintain more than 2 million TCBs at any time. For this reason, some TCP implementations prefer to perform an abrupt connection release by sending a *RST* segment to close the connection [AW05] and immediately discard the corresponding *TCB*. However, if the *RST* segment is lost, the remote host continues to maintain a *TCB* for a connection no longer exists. This optimisation reduces the number of TCBs maintained by the host sending the *RST* segment but at the potential cost of increased processing on the remote host when the *RST* segment is lost.

4.3.3 TCP reliable data transfer

The original TCP data transfer mechanisms were defined in **RFC 793**. Based on the experience of using TCP on the growing global Internet, this part of the TCP specification has been updated and improved several times, always while preserving the backward compatibility with older TCP implementations. In this section, we review the main data transfer mechanisms used by TCP.

TCP is a window-based transport protocol that provides a bi-directional byte stream service. This has several implications on the fields of the TCP header and the mechanisms used by TCP. The three fields of the TCP header are :

- *sequence number*. TCP uses a 32 bits sequence number. The *sequence number* placed in the header of a TCP segment containing data is the sequence number of the first byte of the payload of the TCP segment.
- *acknowledgement number*. TCP uses cumulative positive acknowledgements. Each TCP segment contains the *sequence number* of the next byte that the sender of the acknowledgement expects to receive from the remote host. In theory, the *acknowledgement number* is only valid if the *ACK* flag of the TCP header is set. In practice almost all [15] TCP segments have their *ACK* flag set.
- *window*. a TCP receiver uses this 16 bits field to indicate the current size of its receive window expressed in bytes.

Note: The Transmission Control Block

For each established TCP connection, a TCP implementation must maintain a Transmission Control Block (*TCB*). A TCB contains all the information required to send and receive segments on this connection **RFC 793**. This includes [16] :

- the local IP address
- the remote IP address
- the local TCP port number
- the remote TCP port number
- the current state of the TCP FSM
- the *maximum segment size* (MSS)

[15] In practice, only the *SYN* segment do not have their *ACK* flag set.

[16] A complete TCP implementation contains additional information in its TCB, notably to support the *urgent* pointer. However, this part of TCP is not discussed in this book. Refer to **RFC 793** and **RFC 2140** for more details about the TCB.

- *snd.nxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send uses this sequence number)
- *snd.una* : the earliest sequence number that has been sent but has not yet been acknowledged
- *snd.wnd* : the current size of the sending window (in bytes)
- *rcv.nxt* : the sequence number of the next byte that is expected to be received from the remote host
- *rcv.wnd* : the current size of the receive window advertised by the remote host
- *sending buffer* : a buffer used to store all unacknowledged data
- *receiving buffer* : a buffer to store all data received from the remote host that has not yet been delivered to the user. Data may be stored in the *receiving buffer* because either it was not received in sequence or because the user is too slow to process it

---

The original TCP specification can be categorised as a transport protocol that provides a byte stream service and uses *go-back-n*.

To send new data on an established connection, a TCP entity performs the following operations on the corresponding TCB. It first checks that the *sending buffer* does not contain more data than the receive window advertised by the remote host (*rcv.wnd*). If the window is not full, up to *MSS* bytes of data are placed in the payload of a TCP segment. The *sequence number* of this segment is the sequence number of the first byte of the payload. It is set to the first available sequence number : *snd.nxt* and *snd.nxt* is incremented by the length of the payload of the TCP segment. The *acknowledgement number* of this segment is set to the current value of *rcv.nxt* and the *window* field of the TCP segment is computed based on the current occupancy of the *receiving buffer*. The data is kept in the *sending buffer* in case it needs to be retransmitted later.

When a TCP segment with the *ACK* flag set is received, the following operations are performed. *rcv.wnd* is set to the value of the *window* field of the received segment. The *acknowledgement number* is compared to *snd.una*. The newly acknowledged data is remove from the *sending buffer* and *snd.una* is updated. If the TCP segment contained data, the *sequence number* is compared to *rcv.nxt*. If they are equal, the segment was received in sequence and the data can be delivered to the user and *rcv.nxt* is updated. The contents of the *receiving buffer* is checked to see whether other data already present in this buffer can be delivered in sequence to the user. If so, *rcv.nxt* is updated again. Otherwise, the segment's payload is placed in the *receiving buffer*.

Segment transmission strategies

In a transport protocol such as TCP that offers a bytestream, a practical issue that was left as an implementation choice in **RFC 793** is to decide when a new TCP segment containing data must be sent. There are two simple and extreme implementation choices. The first implementation choice is to send a TCP segment as soon as the user has requested the transmission of some data. This allows TCP to provide a low delay service. However, if the user is sending data one byte at a time, TCP would place each user byte in a segment containing 20 bytes of TCP header [17]. This is a huge overhead that is not acceptable in wide area networks. A second simple solution would be to only transmit a new TCP segment once the user has produced MSS bytes of data. This solution reduces the overhead, but at the cost of a potentially very high delay.

An elegant solution to this problem was proposed by John Nagle in **RFC 896**. John Nagle observed that the overhead caused by the TCP header was a problem in wide area connections, but less in local area connections where the available bandwidth is usually higher. He proposed the following rules to decide to send a new data segment when a new data has been produced by the user or a new ack segment has been received

```
if rcv.wnd>= MSS and len(data) >= MSS :
  send one MSS-sized segment
else
  if there are unacknowledged data:
    place data in buffer until acknowledgement has been received
  else
    send one TCP segment containing all buffered data
```

[17] This TCP segment is then placed in an IP header. We describe IPv4 and IPv6 in the next chapter. The minimum size of the IPv4 (resp. IPv6) header is 20 bytes (resp. 40 bytes).

The first rule ensures that a TCP connection used for bulk data transfer always sends full TCP segments. The second rule sends one partially filled TCP segment every round-trip-time.

This algorithm, called the Nagle algorithm, takes a few lines of code in all TCP implementations. These lines of code have a huge impact on the packets that are exchanged in TCP/IP networks. Researchers have analysed the distribution of the packet sizes by capturing and analysing all the packets passing through a given link. These studies have shown several important results :

- in TCP/IPv4 networks, a large fraction of the packets are TCP segments that contain only an acknowledgement. These packets usually account for 40-50% of the packets passing through the studied link
- in TCP/IPv4 networks, most of the bytes are exchanged in long packets, usually packets containing up to 1460 bytes of payload which is the default MSS for hosts attached to an Ethernet network, the most popular type of LAN

The figure below provides a distribution of the packet sizes measured on a link. It shows a three-modal distribution of the packet size. 50% of the packets contain pure TCP acknowledgements and occupy 40 bytes. About 20% of the packets contain about 500 bytes [18] of user data and 12% of the packets contain 1460 bytes of user data. However, most of the user data is transported in large packets. This packet size distribution has implications on the design of routers as we discuss in the next chapter.

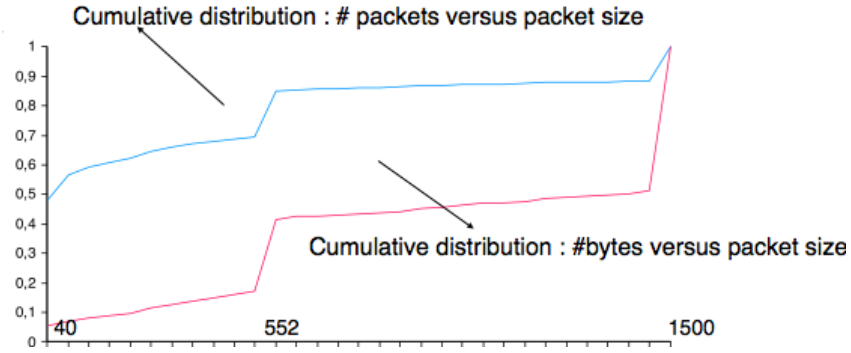

Figure 4.43: Packet size distribution in the Internet

Recent measurements indicate that these packet size distributions are still valid in today's Internet, although the packet distribution tends to become bimodal with small packets corresponding to TCP pure acks (40-64 bytes depending on the utilisation of TCP options) and large 1460-bytes packets carrying most of the user data.

TCP windows

From a performance point of view, one of the main limitations of the original TCP specification is the 16 bits *window* field in the TCP header. As this field indicates the current size of the receive window in bytes, it limits the TCP receive window at 65535 bytes. This limitation was not a severe problem when TCP was designed since at that time high-speed wide area networks offered a maximum bandwidth of 56 kbps. However, in today's network, this limitation is not acceptable anymore. The table below provides the rough [19] maximum throughput that can be achieved by a TCP connection with a 64 KBytes window in function of the connection's round-trip-time

RTT	Maximum Throughput
1 msec	524 Mbps
10 msec	52.4 Mbps
100 msec	5.24 Mbps
500 msec	1.05 Mbps

To solve this problem, a backward compatible extension that allows TCP to use larger receive windows was proposed in **RFC 1323**. Today, most TCP implementations support this option. The basic idea is that instead of

[18] When these measurements were taken, some hosts had a default MSS of 552 bytes (e.g. BSD Unix derivatives) or 536 bytes (the default MSS specified in **RFC 793**). Today, most TCP implementation derive the MSS from the maximum packet size of the LAN interface they use (Ethernet in most cases).

[19] A precise estimation of the maximum bandwidth that can be achieved by a TCP connection should take into account the overhead of the TCP and IP headers as well.

storing *snd.wnd* and *rcv.wnd* as 16 bits integers in the *TCB*, they should be stored as 32 bits integers. As the TCP segment header only contains 16 bits to place the window field, it is impossible to copy the value of *snd.wnd* in each sent TCP segment. Instead the header contains *snd.wnd* >> *S* where *S* is the scaling factor ($0 \leq S \leq 14$) negotiated during connection establishment. The client adds its proposed scaling factor as a TCP option in the *SYN* segment. If the server supports **RFC 1323**, it places in the *SYN+ACK* segment the scaling factor that it uses when advertising its own receive window. The local and remote scaling factors are included in the *TCB*. If the server does not support **RFC 1323**, it ignores the received option and no scaling is applied.

By using the window scaling extensions defined in **RFC 1323**, TCP implementations can use a receive buffer of up to 1 GByte. With such a receive buffer, the maximum throughput that can be achieved by a single TCP connection becomes :

RTT	Maximum Throughput
1 msec	8590 Gbps
10 msec	859 Gbps
100 msec	86 Gbps
500 msec	17 Gbps

These throughputs are acceptable in today's networks. However, there are already servers having 10 Gbps interfaces... Early TCP implementations had fixed receiving and sending buffers [20]. Today's high performance implementations are able to automatically adjust the size of the sending and receiving buffer to better support high bandwidth flows [SMM1998]

TCP's retransmission timeout

In a go-back-n transport protocol such as TCP, the retransmission timeout must be correctly set in order to achieve good performance. If the retransmission timeout expires too early, then bandwidth is wasted by retransmitting segments that have already been correctly received; whereas if the retransmission timeout expires too late, then bandwidth is wasted because the sender is idle waiting for the expiration of its retransmission timeout.

A good setting of the retransmission timeout clearly depends on an accurate estimation of the round-trip-time of each TCP connection. The round-trip-time differs between TCP connections, but may also change during the lifetime of a single connection. For example, the figure below shows the evolution of the round-trip-time between two hosts during a period of 45 seconds.

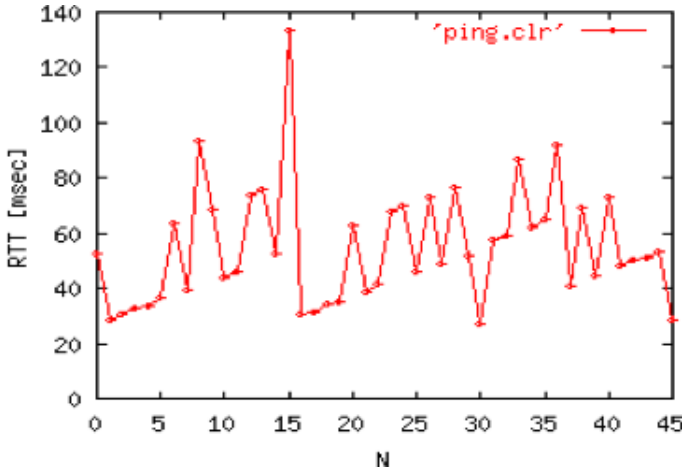

Figure 4.44: Evolution of the round-trip-time between two hosts

The easiest solution to measure the round-trip-time on a TCP connection is to measure the delay between the transmission of a data segment and the reception of a corresponding acknowledgement [21]. As illustrated in the

[20] See http://fasterdata.es.net/tuning.html for more information on how to tune a TCP implementation

[21] In theory, a TCP implementation could store the timestamp of each data segment transmitted and compute a new estimate for the round-trip-time upon reception of the corresponding acknowledgement. However, using such frequent measurements introduces a lot of noise in practice and many implementations still measure the round-trip-time once per round-trip-time by recording the transmission time of one segment at a time **RFC 2988**

figure below, this measurement works well when there are no segment losses.

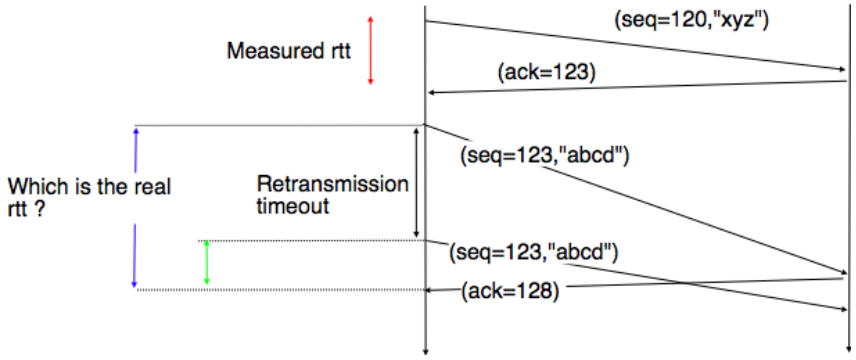

Figure 4.45: How to measure the round-trip-time ?

However, when a data segment is lost, as illustrated in the bottom part of the figure, the measurement is ambiguous as the sender cannot determine whether the received acknowledgement was triggered by the first transmission of segment *123* or its retransmission. Using incorrect round-trip-time estimations could lead to incorrect values of the retransmission timeout. For this reason, Phil Karn and Craig Partridge proposed, in [KP91], to ignore the round-trip-time measurements performed during retransmissions.

To avoid this ambiguity in the estimation of the round-trip-time when segments are retransmitted, recent TCP implementations rely on the *timestamp option* defined in **RFC 1323**. This option allows a TCP sender to place two 32 bit timestamps in each TCP segment that it sends. The first timestamp, TS Value (*TSval*) is chosen by the sender of the segment. It could for example be the current value of its real-time clock [22]. The second value, TS Echo Reply (*TSecr*), is the last *TSval* that was received from the remote host and stored in the *TCB*. The figure below shows how the utilization of this timestamp option allows for the disambiguation of the round-trip-time measurement when there are retransmissions.

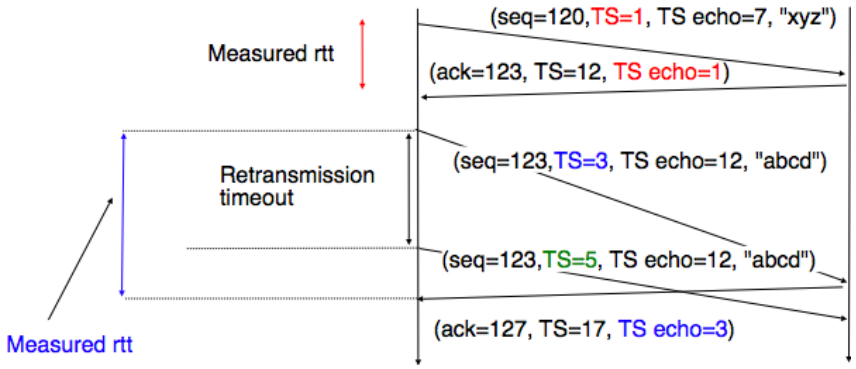

Figure 4.46: Disambiguating round-trip-time measurements with the **RFC 1323** timestamp option

Once the round-trip-time measurements have been collected for a given TCP connection, the TCP entity must compute the retransmission timeout. As the round-trip-time measurements may change during the lifetime of a connection, the retransmission timeout may also change. At the beginning of a connection [23] , the TCP entity that sends a *SYN* segment does not know the round-trip-time to reach the remote host and the initial retransmission timeout is usually set to 3 seconds **RFC 2988**.

The original TCP specification proposed in **RFC 793** to include two additional variables in the TCB :

- *srtt* : the smoothed round-trip-time computed as $srrt = (\alpha \times srtt) + ((1 - \alpha) \times rtt)$ where *rtt* is the round-trip-time measured according to the above procedure and α a smoothing factor (e.g. 0.8 or 0.9)

[22] Some security experts have raised concerns that using the real-time clock to set the *TSval* in the timestamp option can leak information such as the system's up-time. Solutions proposed to solve this problem may be found in [CNPI09]

[23] As a TCP client often establishes several parallel or successive connections with the same server, **RFC 2140** has proposed to reuse for a new connection some information that was collected in the TCB of a previous connection, such as the measured rtt. However, this solution has not been widely implemented.

- *rto* : the retransmission timeout is computed as $rto = min(60, max(1, \beta \times srtt))$ where β is used to take into account the delay variance (value : 1.3 to 2.0). The *60* and *1* constants are used to ensure that the *rto* is not larger than one minute nor smaller than 1 second.

However, in practice, this computation for the retransmission timeout did not work well. The main problem was that the computed *rto* did not correctly take into account the variations in the measured round-trip-time. *Van Jacobson* proposed in his seminal paper [Jacobson1988] an improved algorithm to compute the *rto* and implemented it in the BSD Unix distribution. This algorithm is now part of the TCP standard **RFC 2988**.

Jacobson's algorithm uses two state variables, *srtt* the smoothed *rtt* and *rttvar* the estimation of the variance of the *rtt* and two parameters : α and β. When a TCP connection starts, the first *rto* is set to *3* seconds. When a first estimation of the *rtt* is available, the *srtt*, *rttvar* and *rto* are computed as

```
srtt=rtt
rttvar=rtt/2
rto=srtt+4*rttvar
```

Then, when other rtt measurements are collected, *srtt* and *rttvar* are updated as follows :

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rto = srtt + 4 \times rttvar$$

The proposed values for the parameters are $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$. This allows a TCP implementation, implemented in the kernel, to perform the *rtt* computation by using shift operations instead of the more costly floating point operations [Jacobson1988]. The figure below illustrates the computation of the *rto* upon *rtt* changes.

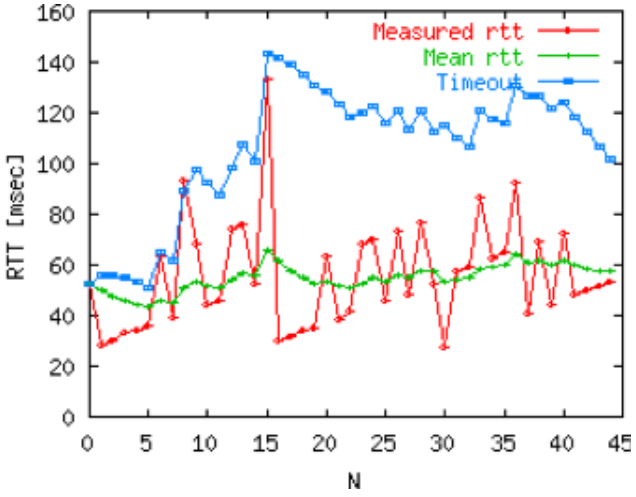

Figure 4.47: Example computation of the *rto*

Advanced retransmission strategies

The default go-back-n retransmission strategy was defined in **RFC 793**. When the retransmission timer expires, TCP retransmits the first unacknowledged segment (i.e. the one having sequence number *snd.una*). After each expiration of the retransmission timeout, **RFC 2988** recommends to double the value of the retransmission timeout. This is called an *exponential backoff*. This doubling of the retransmission timeout after a retransmission was included in TCP to deal with issues such as network/receiver overload and incorrect initial estimations of the retransmission timeout. If the same segment is retransmitted several times, the retransmission timeout is doubled after every retransmission until it reaches a configured maximum. **RFC 2988** suggests a maximum retransmission timeout of at least 60 seconds. Once the retransmission timeout reaches this configured maximum, the remote host is considered to be unreachable and the TCP connection is closed.

This retransmission strategy has been refined based on the experience of using TCP on the Internet. The first refinement was a clarification of the strategy used to send acknowledgements. As TCP uses piggybacking, the

easiest and less costly method to send acknowledgements is to place them in the data segments sent in the other direction. However, few application layer protocols exchange data in both directions at the same time and thus this method rarely works. For an application that is sending data segments in one direction only, the remote TCP entity returns empty TCP segments whose only useful information is their acknowledgement number. This may cause a large overhead in wide area network if a pure *ACK* segment is sent in response to each received data segment. Most TCP implementations use a *delayed acknowledgement* strategy. This strategy ensures that piggybacking is used whenever possible, otherwise pure *ACK* segments are sent for every second received data segments when there are no losses. When there are losses or reordering, *ACK* segments are more important for the sender and they are sent immediately **RFC 813 RFC 1122**. This strategy relies on a new timer with a short delay (e.g. 50 milliseconds) and one additional flag in the TCB. It can be implemented as follows

```
reception of a data segment:
   if pkt.seq==rcv.nxt:   # segment received in sequence
      if delayedack :
         send pure ack segment
         cancel acktimer
         delayedack=False
      else:
         delayedack=True
         start acktimer
   else:                       # out of sequence segment
      send pure ack segment
      if delayedack:
         delayedack=False
         cancel acktimer

transmission of a data segment:  # piggyback ack
   if delayedack:
      delayedack=False
      cancel acktimer

acktimer expiration:
   send pure ack segment
   delayedack=False
```

Due to this delayed acknowledgement strategy, during a bulk transfer, a TCP implementation usually acknowledges every second TCP segment received.

The default go-back-n retransmission strategy used by TCP has the advantage of being simple to implement, in particular on the receiver side, but when there are losses, a go-back-n strategy provides a lower performance than a selective repeat strategy. The TCP developers have designed several extensions to TCP to allow it to use a selective repeat strategy while maintaining backward compatibility with older TCP implementations. These TCP extensions assume that the receiver is able to buffer the segments that it receives out-of-sequence.

The first extension that was proposed is the fast retransmit heuristic. This extension can be implemented on TCP senders and thus does not require any change to the protocol. It only assumes that the TCP receiver is able to buffer out-of-sequence segments.

From a performance point of view, one issue with TCP's *retransmission timeout* is that when there are isolated segment losses, the TCP sender often remains idle waiting for the expiration of its retransmission timeouts. Such isolated losses are frequent in the global Internet [Paxson99]. A heuristic to deal with isolated losses without waiting for the expiration of the retransmission timeout has been included in many TCP implementations since the early 1990s. To understand this heuristic, let us consider the figure below that shows the segments exchanged over a TCP connection when an isolated segment is lost.

As shown above, when an isolated segment is lost the sender receives several *duplicate acknowledgements* since the TCP receiver immediately sends a pure acknowledgement when it receives an out-of-sequence segment. A duplicate acknowledgement is an acknowledgement that contains the same *acknowledgement number* as a previous segment. A single duplicate acknowledgement does not necessarily imply that a segment was lost, as a simple reordering of the segments may cause duplicate acknowledgements as well. Measurements [Paxson99] have shown that segment reordering is frequent in the Internet. Based on these observations, the *fast retransmit* heuristic has been included in most TCP implementations. It can be implemented as follows

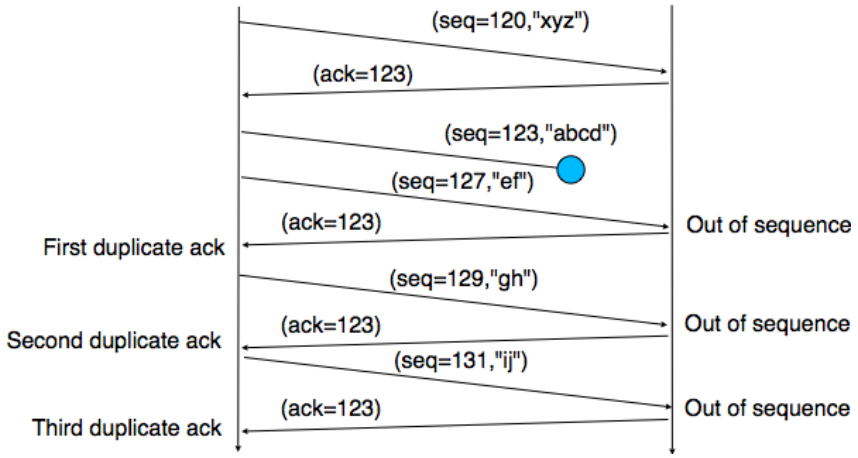

Figure 4.48: Detecting isolated segment losses

```
ack arrival:
    if tcp.ack==snd.una:    # duplicate acknowledgement
       dupacks++
       if dupacks==3:
          retransmit segment(snd.una)
    else:
       dupacks=0
       # process acknowledgement
```

This heuristic requires an additional variable in the TCB (*dupacks*). Most implementations set the default number of duplicate acknowledgements that trigger a retransmission to 3. It is now part of the standard TCP specification **RFC 2581**. The *fast retransmit* heuristic improves the TCP performance provided that isolated segments are lost and the current window is large enough to allow the sender to send three duplicate acknowledgements.

The figure below illustrates the operation of the *fast retransmit* heuristic.

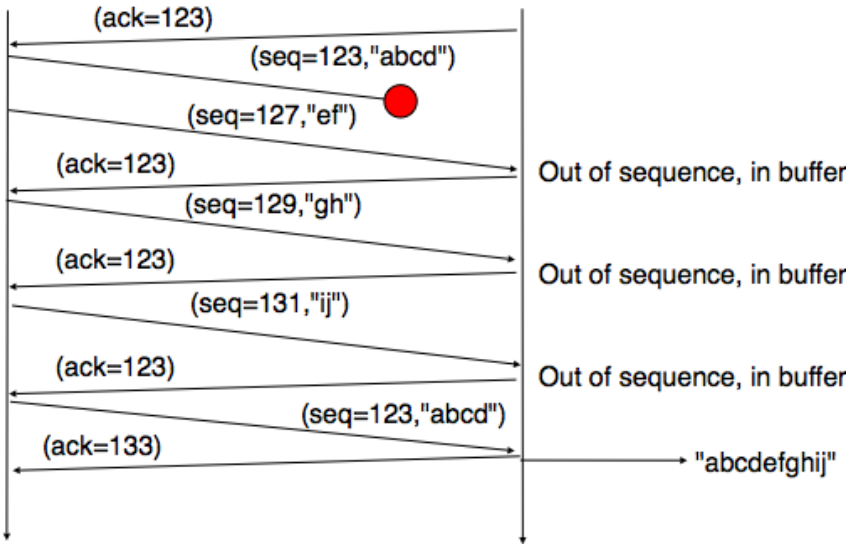

Figure 4.49: TCP fast retransmit heuristics

When losses are not isolated or when the windows are small, the performance of the *fast retransmit* heuristic decreases. In such environments, it is necessary to allow a TCP sender to use a selective repeat strategy instead of the default go-back-n strategy. Implementing selective-repeat requires a change to the TCP protocol as the receiver needs to be able to inform the sender of the out-of-order segments that it has already received. This can be done by using the Selective Acknowledgements (SACK) option defined in **RFC 2018**. This TCP option is

negotiated during the establishment of a TCP connection. If both TCP hosts support the option, SACK blocks can be attached by the receiver to the segments that it sends. SACK blocks allow a TCP receiver to indicate the blocks of data that it has received correctly but out of sequence. The figure below illustrates the utilisation of the SACK blocks.

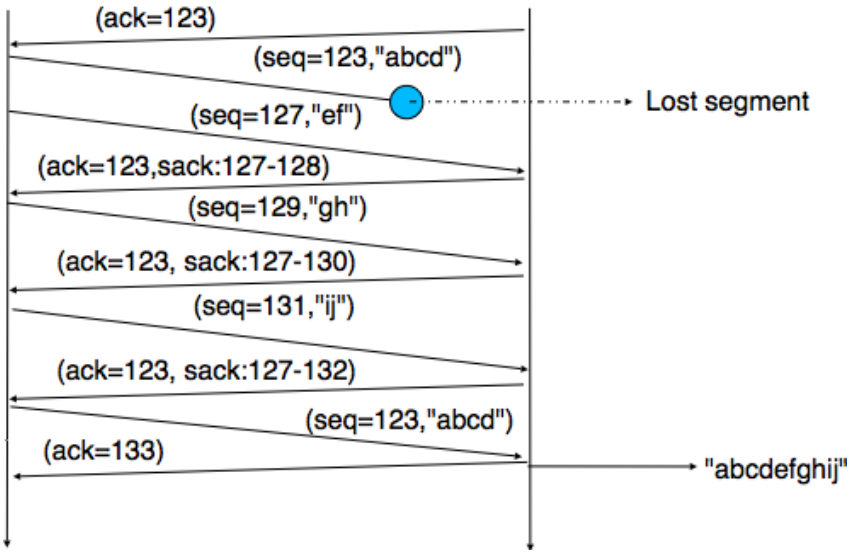

Figure 4.50: TCP selective acknowledgements

An SACK option contains one or more blocks. A block corresponds to all the sequence numbers between the *left edge* and the *right edge* of the block. The two edges of the block are encoded as 32 bit numbers (the same size as the TCP sequence number) in an SACK option. As the SACK option contains one byte to encode its type and one byte for its length, a SACK option containing b blocks is encoded as a sequence of $2 + 8 \times b$ bytes. In practice, the size of the SACK option can be problematic as the optional TCP header extension cannot be longer than 44 bytes. As the SACK option is usually combined with the **RFC 1323** timestamp extension, this implies that a TCP segment cannot usually contain more than three SACK blocks. This limitation implies that a TCP receiver cannot always place in the SACK option that it sends, information about all the received blocks.

To deal with the limited size of the SACK option, a TCP receiver currently having more than 3 blocks inside its receiving buffer must select the blocks to place in the SACK option. A good heuristic is to put in the SACK option the blocks that have most recently changed, as the sender is likely to be already aware of the older blocks.

When a sender receives an SACK option indicating a new block and thus a new possible segment loss, it usually does not retransmit the missing segment(s immediately. To deal with reordering, a TCP sender can use a heuristic similar to *fast retransmit* by retransmitting a gap only once it has received three SACK options indicating this gap. It should be noted that the SACK option does not supersede the *acknowledgement number* of the TCP header. A TCP sender can only remove data from its sending buffer once they have been acknowledged by TCP's cumulative acknowledgements. This design was chosen for two reasons. First, it allows the receiver to discard parts of its receiving buffer when it is running out of memory without loosing data. Second, as the SACK option is not transmitted reliably, the cumulative acknowledgements are still required to deal with losses of *ACK* segments carrying only SACK information. Thus, the SACK option only serves as a hint to allow the sender to optimise its retransmissions.

TCP congestion control

In the previous sections, we have explained the mechanisms that TCP uses to deal with transmission errors and segment losses. In a heterogeneous network such as the Internet or enterprise IP networks, endsystems have very different levels of performance. Some endsystems are high-end servers attached to 10 Gbps links while others are mobile devices attached to a very low bandwidth wireless link. Despite these huge differences in performance, a mobile device should be able to efficiently exchange segments with a high-end server.

To understand this problem better, let us consider the scenario shown in the figure below, where a server (A)

attached to a *10 Mbps* link is sending TCP segments to another computer (*C*) through a path that contains a *2 Mbps* link.

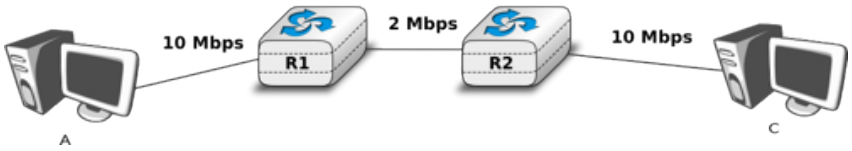

Figure 4.51: TCP over heterogeneous links

In this network, the TCP segments sent by the server reach router *R1*. *R1* forwards the segments towards router *R2*. Router *R2* can potentially receive segments at *10 Mbps*, but it can only forward them at *2 Mbps* to router *R2* and then to host *C*. Router *R2* contains buffers that allow it to store the packets that cannot immediately be forwarded to their destination. To understand the operation of TCP in this environment, let us consider a simplified model of this network where host *A* is attached to a *10 Mbps* link to a queue that represents the buffers of router *R2*. This queue is emptied at a rate of *2 Mbps*.

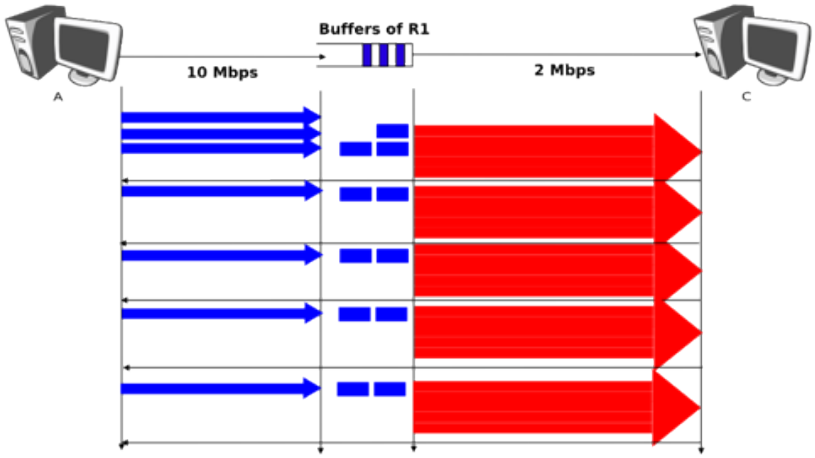

Figure 4.52: TCP self clocking

Let us consider that host *A* uses a window of three segments. It thus sends three back-to-back segments at *10 Mbps* and then waits for an acknowledgement. Host *A* stops sending segments when its window is full. These segments reach the buffers of router *R2*. The first segment stored in this buffer is sent by router *R2* at a rate of *2 Mbps* to the destination host. Upon reception of this segment, the destination sends an acknowledgement. This acknowledgement allows host *A* to transmit a new segment. This segment is stored in the buffers of router *R2* while it is transmitting the second segment that was sent by host *A*... Thus, after the transmission of the first window of segments, TCP sends one data segment after the reception of each acknowledgement returned by the destination [24] . In practice, the acknowledgements sent by the destination serve as a kind of *clock* that allows the sending host to adapt its transmission rate to the rate at which segments are received by the destination. This *TCP self-clocking* is the first mechanism that allows TCP to adapt to heterogeneous networks [Jacobson1988]. It depends on the availability of buffers to store the segments that have been sent by the sender but have not yet been transmitted to the destination.

However, TCP is not always used in this environment. In the global Internet, TCP is used in networks where a large number of hosts send segments to a large number of receivers. For example, let us consider the network

[24] If the destination is using delayed acknowledgements, the sending host sends two data segments after each acknowledgement.

depicted below which is similar to the one discussed in [Jacobson1988] and **RFC 896**. In this network, we assume that the buffers of the router are infinite to ensure that no packet is lost.

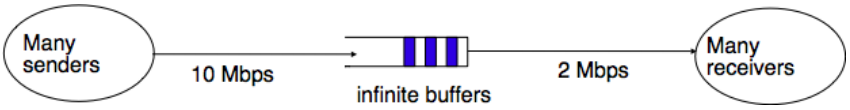

Figure 4.53: The congestion collapse problem

If many TCP senders are attached to the left part of the network above, they all send a window full of segments. These segments are stored in the buffers of the router before being transmitted towards their destination. If there are many senders on the left part of the network, the occupancy of the buffers quickly grows. A consequence of the buffer occupancy is that the round-trip-time, measured by TCP, between the sender and the receiver increases. Consider a network where 10,000 bits segments are sent. When the buffer is empty, such a segment requires 1 millisecond to be transmitted on the *10 Mbps* link and 5 milliseconds to be the transmitted on the *2 Mbps* link. Thus, the round-trip-time measured by TCP is roughly 6 milliseconds if we ignore the propagation delay on the links. Most routers manage their buffers as a FIFO queue [25]. If the buffer contains 100 segments, the round-trip-time becomes $1 + 100 \times 5 + 5$ milliseconds as new segments are only transmitted on the *2 Mbps* link once all previous segments have been transmitted. Unfortunately, TCP uses a retransmission timer and performs *go-back-n* to recover from transmission errors. If the buffer occupancy is high, TCP assumes that some segments have been lost and retransmits a full window of segments. This increases the occupancy of the buffer and the delay through the buffer... Furthermore, the buffer may store and send on the low bandwidth links several retransmissions of the same segment. This problem is called *congestion collapse*. It occurred several times in the late 1980s. For example, [Jacobson1988] notes that in 1986, the usable bandwidth of a 32 Kbits link dropped to 40 bits per second due to congestion collapse [26] !

The *congestion collapse* is a problem that all heterogeneous networks face. Different mechanisms have been proposed in the scientific literature to avoid or control network congestion. Some of them have been implemented and deployed in real networks. To understand this problem in more detail, let us first consider a simple network with two hosts attached to a high bandwidth link that are sending segments to destination *C* attached to a low bandwidth link as depicted below.

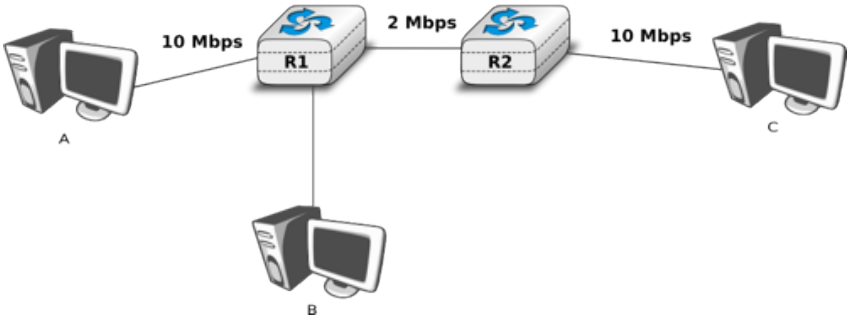

Figure 4.54: The congestion problem

To avoid *congestion collapse*, the hosts must regulate their transmission rate [27] by using a *congestion control* mechanism. Such a mechanism can be implemented in the transport layer or in the network layer. In TCP/IP networks, it is implemented in the transport layer, but other technologies such as *Asynchronous Transfer Mode (ATM)* or *Frame Relay* include congestion control mechanisms in lower layers.

Let us first consider the simple problem of a set of i hosts that share a single bottleneck link as shown in the example above. In this network, the congestion control scheme must achieve the following objectives [CJ1989] :

[25] We discuss in another chapter other possible organisations of the router's buffers.

[26] At this time, TCP implementations were mainly following **RFC 791**. The round-trip-time estimations and the retransmission mechanisms were very simple. TCP was improved after the publication of [Jacobson1988]

[27] In this section, we focus on congestion control mechanisms that regulate the transmission rate of the hosts. Other types of mechanisms have been proposed in the literature. For example, *credit-based* flow-control has been proposed to avoid congestion in ATM networks [KR1995]. With a credit-based mechanism, hosts can only send packets once they have received credits from the routers and the credits depend on the occupancy of the router's buffers.

1. The congestion control scheme must *avoid congestion*. In practice, this means that the bottleneck link cannot be overloaded. If $r_i(t)$ is the transmission rate allocated to host i at time t and R the bandwidth of the bottleneck link, then the congestion control scheme should ensure that, on average, $\forall t \sum r_i(t) \leq R$.
2. The congestion control scheme must be *efficient*. The bottleneck link is usually both a shared and an expensive resource. Usually, bottleneck links are wide area links that are much more expensive to upgrade than the local area networks. The congestion control scheme should ensure that such links are efficiently used. Mathematically, the control scheme should ensure that $\forall t \sum r_i(t) \approx R$.
3. The congestion control scheme should be *fair*. Most congestion schemes aim at achieving *max-min fairness*. An allocation of transmission rates to sources is said to be *max-min fair* if :

- no link in the network is congested
- the rate allocated to source j cannot be increased without decreasing the rate allocated to a source i whose allocation is smaller than the rate allocated to source j [Leboudec2008] .

Depending on the network, a *max-min fair allocation* may not always exist. In practice, *max-min fairness* is an ideal objective that cannot necessarily be achieved. When there is a single bottleneck link as in the example above, *max-min fairness* implies that each source should be allocated the same transmission rate.

To visualise the different rate allocations, it is useful to consider the graph shown below. In this graph, we plot on the *x-axis* (resp. *y-axis*) the rate allocated to host *B* (resp. *A*). A point in the graph (r_B, r_A) corresponds to a possible allocation of the transmission rates. Since there is a *2 Mbps* bottleneck link in this network, the graph can be divided into two regions. The lower left part of the graph contains all allocations (r_B, r_A) such that the bottleneck link is not congested ($r_A + r_B < 2$). The right border of this region is the *efficiency line*, i.e. the set of allocations that completely utilise the bottleneck link ($r_A + r_B = 2$). Finally, the *fairness line* is the set of fair allocations.

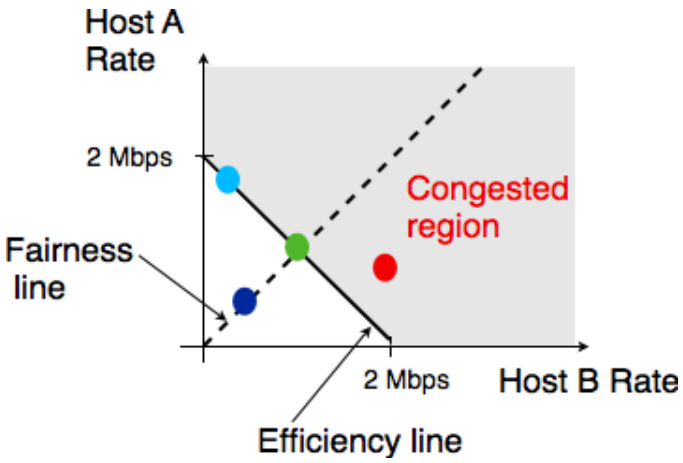

Figure 4.55: Possible allocated transmission rates

As shown in the graph above, a rate allocation may be fair but not efficient (e.g. $r_A = 0.7, r_B = 0.7$), fair and efficient (e.g. $r_A = 1, r_B = 1$) or efficient but not fair (e.g. $r_A = 1.5, r_B = 0.5$). Ideally, the allocation should be both fair and efficient. Unfortunately, maintaining such an allocation with fluctuations in the number of flows that use the network is a challenging problem. Furthermore, there might be several thousands of TCP connections or more that pass through the same link [28].

To deal with these fluctuations in demand, which result in fluctuations in the available bandwidth, computer networks use a congestion control scheme. This congestion control scheme should achieve the three objectives listed above. Some congestion control schemes rely on a close cooperation between the endhosts and the routers, while others are mainly implemented on the endhosts with limited support from the routers.

[28] For example, the measurements performed in the Sprint network in 2004 reported more than 10k active TCP connections on a link, see https://research.sprintlabs.com/packstat/packetoverview.php. More recent information about backbone links may be obtained from caida 's realtime measurements, see e.g. http://www.caida.org/data/realtime/passive/

A congestion control scheme can be modelled as an algorithm that adapts the transmission rate ($r_i(t)$) of host i based on the feedback received from the network. Different types of feedbacks are possible. The simplest scheme is a binary feedback [CJ1989] [Jacobson1988] where the hosts simply learn whether the network is congested or not. Some congestion control schemes allow the network to regularly send an allocated transmission rate in Mbps to each host [BF1995].

Let us focus on the binary feedback scheme which is the most widely used today. Intuitively, the congestion control scheme should decrease the transmission rate of a host when congestion has been detected in the network, in order to avoid congestion collapse. Furthermore, the hosts should increase their transmission rate when the network is not congested. Otherwise, the hosts would not be able to efficiently utilise the network. The rate allocated to each host fluctuates with time, depending on the feedback received from the network. The figure below illustrates the evolution of the transmission rates allocated to two hosts in our simple network. Initially, two hosts have a low allocation, but this is not efficient. The allocations increase until the network becomes congested. At this point, the hosts decrease their transmission rate to avoid congestion collapse. If the congestion control scheme works well, after some time the allocations should become both fair and efficient.

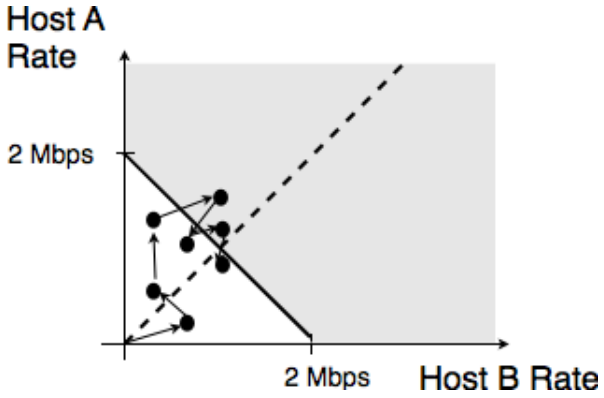

Figure 4.56: Evolution of the transmission rates

Various types of rate adaption algorithms are possible. Dah Ming Chiu and Raj Jain have analysed, in [CJ1989], different types of algorithms that can be used by a source to adapt its transmission rate to the feedback received from the network. Intuitively, such a rate adaptation algorithm increases the transmission rate when the network is not congested (ensure that the network is efficiently used) and decrease the transmission rate when the network is congested (to avoid congestion collapse).

The simplest form of feedback that the network can send to a source is a binary feedback (the network is congested or not congested). In this case, a *linear* rate adaptation algorithm can be expressed as :

- $rate(t+1) = \alpha_C + \beta_C rate(t)$ when the network is congested
- $rate(t+1) = \alpha_N + \beta_N rate(t)$ when the network is *not* congested

With a linear adaption algorithm, $\alpha_C, \alpha_N, \beta_C$ and β_N are constants. The analysis of [CJ1989] shows that to be fair and efficient, such a binary rate adaption mechanism must rely on *Additive Increase and Multiplicative Decrease*. When the network is not congested, the hosts should slowly increase their transmission rate ($\beta_N = 1 \; and \; \alpha_N > 0$). When the network is congested, the hosts must multiplicatively decrease their transmission rate ($\beta_C < 1 \; and \; \alpha_C = 0$). Such an AIMD rate adaptation algorithm can be implemented by the pseudo-code below

```
# Additive Increase Multiplicative Decrease
if congestion :
   rate=rate*betaC    # multiplicative decrease, betaC<1
else
   rate=rate+alphaN    # additive increase, v0>0
```

Note: Which binary feedback ?

Two types of binary feedback are possible in computer networks. A first solution is to rely on implicit feedback. This is the solution chosen for TCP. TCP's congestion control scheme [Jacobson1988] does not require any cooperation from the router. It only assumes that they use buffers and that they discard packets when there is congestion.

TCP uses the segment losses as an indication of congestion. When there are no losses, the network is assumed to be not congested. This implies that congestion is the main cause of packet losses. This is true in wired networks, but unfortunately not always true in wireless networks. Another solution is to rely on explicit feedback. This is the solution proposed in the DECBit congestion control scheme [RJ1995] and used in Frame Relay and ATM networks. This explicit feedback can be implemented in two ways. A first solution would be to define a special message that could be sent by routers to hosts when they are congested. Unfortunately, generating such messages may increase the amount of congestion in the network. Such a congestion indication packet is thus discouraged **RFC 1812**. A better approach is to allow the intermediate routers to indicate, in the packets that they forward, their current congestion status. Binary feedback can be encoded by using one bit in the packet header. With such a scheme, congested routers set a special bit in the packets that they forward while non-congested routers leave this bit unmodified. The destination host returns the congestion status of the network in the acknowledgements that it sends. Details about such a solution in IP networks may be found in **RFC 3168**. Unfortunately, as of this writing, this solution is still not deployed despite its potential benefits.

The TCP congestion control scheme was initially proposed by Van Jacobson in [Jacobson1988]. The current specification may be found in **RFC 5681**. TCP relies on *Additive Increase and Multiplicative Decrease (AIMD)*. To implement *AIMD*, a TCP host must be able to control its transmission rate. A first approach would be to use timers and adjust their expiration times in function of the rate imposed by *AIMD*. Unfortunately, maintaining such timers for a large number of TCP connections can be difficult. Instead, Van Jacobson noted that the rate of TCP congestion can be artificially controlled by constraining its sending window. A TCP connection cannot send data faster than $\frac{window}{rtt}$ where $window$ is the maximum between the host's sending window and the window advertised by the receiver.

TCP's congestion control scheme is based on a *congestion window*. The current value of the congestion window (*cwnd*) is stored in the TCB of each TCP connection and the window that can be used by the sender is constrained by $min(cwnd, rwin, swin)$ where $swin$ is the current sending window and $rwin$ the last received receive window. The *Additive Increase* part of the TCP congestion control increments the congestion window by *MSS* bytes every round-trip-time. In the TCP literature, this phase is often called the *congestion avoidance* phase. The *Multiplicative Decrease* part of the TCP congestion control divides the current value of the congestion window once congestion has been detected.

When a TCP connection begins, the sending host does not know whether the part of the network that it uses to reach the destination is congested or not. To avoid causing too much congestion, it must start with a small congestion window. [Jacobson1988] recommends an initial window of MSS bytes. As the additive increase part of the TCP congestion control scheme increments the congestion window by MSS bytes every round-trip-time, the TCP connection may have to wait many round-trip-times before being able to efficiently use the available bandwidth. This is especially important in environments where the $bandwidth \times rtt$ product is high. To avoid waiting too many round-trip-times before reaching a congestion window that is large enough to efficiently utilise the network, the TCP congestion control scheme includes the *slow-start* algorithm. The objective of the TCP *slow-start* is to quickly reach an acceptable value for the *cwnd*. During *slow-start*, the congestion window is doubled every round-trip-time. The *slow-start* algorithm uses an additional variable in the TCB : *sshtresh* (*slow-start threshold*). The *ssthresh* is an estimation of the last value of the *cwnd* that did not cause congestion. It is initialised at the sending window and is updated after each congestion event.

In practice, a TCP implementation considers the network to be congested once its needs to retransmit a segment. The TCP congestion control scheme distinguishes between two types of congestion :

- *mild congestion*. TCP considers that the network is lightly congested if it receives three duplicate acknowledgements and performs a fast retransmit. If the fast retransmit is successful, this implies that only one segment has been lost. In this case, TCP performs multiplicative decrease and the congestion window is divided by *2*. The slow-start threshold is set to the new value of the congestion window.
- *severe congestion*. TCP considers that the network is severely congested when its retransmission timer expires. In this case, TCP retransmits the first segment, sets the slow-start threshold to 50% of the congestion window. The congestion window is reset to its initial value and TCP performs a slow-start.

The figure below illustrates the evolution of the congestion window when there is severe congestion. At the beginning of the connection, the sender performs *slow-start* until the first segments are lost and the retransmission timer expires. At this time, the *ssthresh* is set to half of the current congestion window and the congestion window is reset at one segment. The lost segments are retransmitted as the sender again performs slow-start until the

congestion window reaches the *sshtresh*. It then switches to congestion avoidance and the congestion window increases linearly until segments are lost and the retransmission timer expires ...

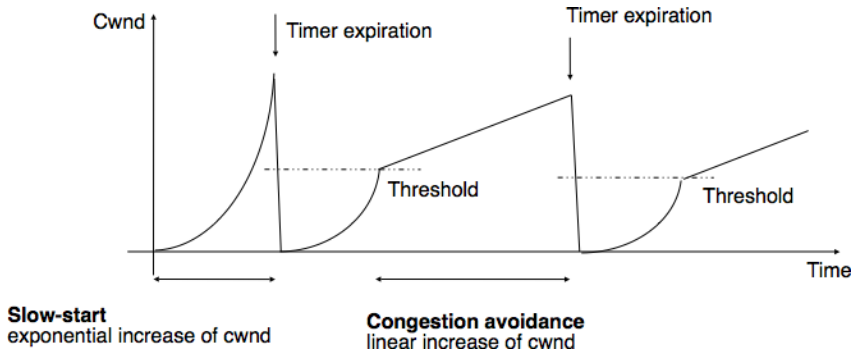

Figure 4.57: Evaluation of the TCP congestion window with severe congestion

The figure below illustrates the evolution of the congestion window when the network is lightly congested and all lost segments can be retransmitted using fast retransmit. The sender begins with a slow-start. A segment is lost but successfully retransmitted by a fast retransmit. The congestion window is divided by 2 and the sender immediately enters congestion avoidance as this was a mild congestion.

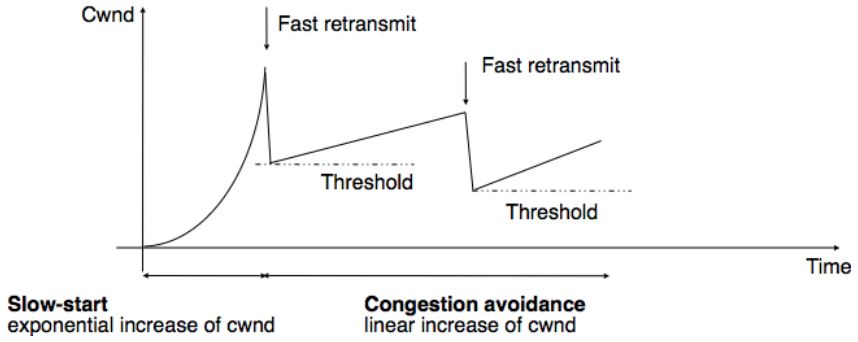

Figure 4.58: Evaluation of the TCP congestion window when the network is lightly congested

Most TCP implementations update the congestion window when they receive an acknowledgement. If we assume that the receiver acknowledges each received segment and the sender only sends MSS sized segments, the TCP congestion control scheme can be implemented using the simplified pseudo-code [29] below

```
# Initialisation
cwnd = MSS;
ssthresh= swin;

# Ack arrival
if tcp.ack > snd.una :  # new ack, no congestion
   if  cwnd < ssthresh :
     # slow-start : increase quickly cwnd
     # double cwnd  every rtt
     cwnd = cwnd + MSS
   else:
     # congestion avoidance : increase slowly cwnd
     # increase cwnd by one mss every rtt
     cwnd = cwnd+ mss*(mss/cwnd)
else: # duplicate or old ack
   if tcp.ack==snd.una:    # duplicate acknowledgement
     dupacks++
     if dupacks==3:
```

[29] In this pseudo-code, we assume that TCP uses unlimited sequence and acknowledgement numbers. Furthermore, we do not detail how the *cwnd* is adjusted after the retransmission of the lost segment by fast retransmit. Additional details may be found in RFC 5681.

```
        retransmitsegment(snd.una)
        ssthresh=max(cwnd/2,2*MSS)
        cwnd=ssthresh
      else:
        dupacks=0
        # ack for old segment, ignored

Expiration of the retransmission timer:
 send(snd.una)      # retransmit first lost segment
 sshtresh=max(cwnd/2,2*MSS)
 cwnd=MSS
```

Furthermore when a TCP connection has been idle for more than its current retransmission timer, it should reset its congestion window to the congestion window size that it uses when the connection begins, as it no longer knows the current congestion state of the network.

Note: Initial congestion window

The original TCP congestion control mechanism proposed in [Jacobson1988] recommended that each TCP connection should begin by setting $cwnd = MSS$. However, in today's higher bandwidth networks, using such a small initial congestion window severely affects the performance for short TCP connections, such as those used by web servers. Since the publication of **RFC 3390**, TCP hosts are allowed to use an initial congestion window of about 4 KBytes, which corresponds to 3 segments in many environments.

Thanks to its congestion control scheme, TCP adapts its transmission rate to the losses that occur in the network. Intuitively, the TCP transmission rate decreases when the percentage of losses increases. Researchers have proposed detailed models that allow the prediction of the throughput of a TCP connection when losses occur [MSMO1997] . To have some intuition about the factors that affect the performance of TCP, let us consider a very simple model. Its assumptions are not completely realistic, but it gives us good intuition without requiring complex mathematics.

This model considers a hypothetical TCP connection that suffers from equally spaced segment losses. If p is the segment loss ratio, then the TCP connection successfully transfers $\frac{1}{p} - 1$ segments and the next segment is lost. If we ignore the slow-start at the beginning of the connection, TCP in this environment is always in congestion avoidance as there are only isolated losses that can be recovered by using fast retransmit. The evolution of the congestion window is thus as shown in the figure below. Note the that *x-axis* of this figure represents time measured in units of one round-trip-time, which is supposed to be constant in the model, and the *y-axis* represents the size of the congestion window measured in MSS-sized segments.

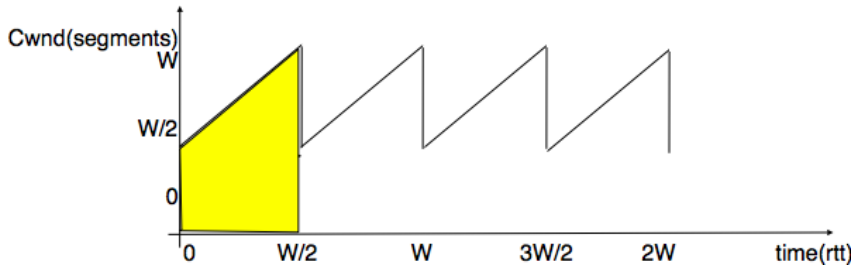

Figure 4.59: Evolution of the congestion window with regular losses

As the losses are equally spaced, the congestion window always starts at some value ($\frac{W}{2}$), and is incremented by one MSS every round-trip-time until it reaches twice this value (W). At this point, a segment is retransmitted and the cycle starts again. If the congestion window is measured in MSS-sized segments, a cycle lasts $\frac{W}{2}$ round-trip-times. The bandwidth of the TCP connection is the number of bytes that have been transmitted during a given period of time. During a cycle, the number of segments that are sent on the TCP connection is equal to the area of the yellow trapeze in the figure. Its area is thus :

$$area = (\frac{W}{2})^2 + \frac{1}{2} \times (\frac{W}{2})^2 = \frac{3 \times W^2}{8}$$

However, given the regular losses that we consider, the number of segments that are sent between two losses (i.e. during a cycle) is by definition equal to $\frac{1}{p}$. Thus, $W = \sqrt{\frac{8}{3 \times p}} = \frac{k}{\sqrt{p}}$. The throughput (in bytes per second) of the TCP connection is equal to the number of segments transmitted divided by the duration of the cycle :

$$Throughput = \frac{area \times MSS}{time} = \frac{\frac{3 \times W^2}{8}}{\frac{W}{2} \times rtt}$$ or, after having eliminated *W*, $Throughput = \sqrt{\frac{3}{2}} \times \frac{MSS}{rtt \times \sqrt{p}}$

More detailed models and the analysis of simulations have shown that a first order model of the TCP throughput when losses occur was $Throughput \approx \frac{k \times MSS}{rtt \times \sqrt{p}}$. This is an important result which shows that :

- TCP connections with a small round-trip-time can achieve a higher throughput than TCP connections having a longer round-trip-time when losses occur. This implies that the TCP congestion control scheme is not completely fair since it favors the connections that have the shorter round-trip-time
- TCP connections that use a large MSS can achieve a higher throughput that the TCP connections that use a shorter MSS. This creates another source of unfairness between TCP connections. However, it should be noted that today most hosts are using almost the same MSS, roughly 1460 bytes.

In general, the maximum throughput that can be achieved by a TCP connection depends on its maximum window size and the round-trip-time if there are no losses. If there are losses, it depends on the MSS, the round-trip-time and the loss ratio.

$$Throughput < min(\frac{window}{rtt}, \frac{k \times MSS}{rtt \times \sqrt{p}})$$

Note: The TCP congestion control zoo

The first TCP congestion control scheme was proposed by Van Jacobson in [Jacobson1988]. In addition to writing the scientific paper, Van Jacobson also implemented the slow-start and congestion avoidance schemes in release 4.3 *Tahoe* of the BSD Unix distributed by the University of Berkeley. Later, he improved the congestion control by adding the fast retransmit and the fast recovery mechanisms in the *Reno* release of 4.3 BSD Unix. Since then, many researchers have proposed, simulated and implemented modifications to the TCP congestion control scheme. Some of these modifications are still used today, e.g. :

- *NewReno* (**RFC 3782**), which was proposed as an improvement of the fast recovery mechanism in the *Reno* implementation
- *TCP Vegas*, which uses changes in the round-trip-time to estimate congestion in order to avoid it [BOP1994]
- *CUBIC*, which was designed for high bandwidth links and is the default congestion control scheme in the Linux 2.6.19 kernel [HRX2008]
- *Compound TCP*, which was designed for high bandwidth links is the default congestion control scheme in several Microsoft operating systems [STBT2009]

A search of the scientific literature will probably reveal more than 100 different variants of the TCP congestion control scheme. Most of them have only been evaluated by simulations. However, the TCP implementation in the recent Linux kernels supports several congestion control schemes and new ones can be easily added. We can expect that new TCP congestion control schemes will always continue to appear.

4.4 Summary

In this chapter, we have studied the transport layer. This layer provides two types of services to the application layer. The unreliable connectionless service is the simplest service offered to applications. On the Internet, this is the service offered by UDP. However, most applications prefer to use a reliable and connection-oriented transport service. We have shown that providing this service was much more complex than providing an unreliable service as the transport layer needs to recover from the errors that occur in the network layer. For this, transport layer protocols rely on several mechanisms. First, they use a handshake mechanism, such as the three-way handshake mechanism, to correctly establish a transport connection. Once the connection has been established, transport entities exchange segments. Each segment contains a sequence number, and the transport layer uses acknowledgements to confirm the segments that have been correctly received. In addition, timers are used to recover from segment losses and sliding windows are used to avoid overflowing the buffers of the transport entities. Finally,

we explained how a transport connection can be safely released. We then discussed the mechanisms that are used in TCP, the reliable transport protocol, used by most applications on the Internet. Most notably, we described the congestion control mechanism that has been included in TCP since the late 1980s and explained how the reliability mechanisms used by TCP have been tuned over the years.

4.5 Exercises

This section is divided in two parts. The first part contains exercises on the principles of transport protocols, including TCP. The second part contains programming challenges packet analysis tools to observe the behaviour of transport protocols.

4.5.1 Principles

1. Consider the Alternating Bit Protocol as described in this chapter

- How does the protocol recover from the loss of a data segment ?
- How does the protocol recovers from the loss of an acknowledgement ?

2. A student proposed to optimise the Alternating Bit Protocol by adding a negative acknowledgment, i.e. the receiver sends a *NAK* control segment when it receives a corrupted data segment. What kind of information should be placed in this control segment and how should the sender react when receiving such a *NAK* ?

3. Transport protocols rely on different types of checksums to verify whether segments have been affected by transmission errors. The most frequently used checksums are :

   - the Internet checksum used by UDP, TCP and other Internet protocols which is defined in **RFC 1071** and implemented in various modules, e.g. http://ilab.cs.byu.edu/cs460/code/ftp/ichecksum.py for a python implementation
   - the 16 bits or the 32 bits Cyclical Redundancy Checks (CRC) that are often used on disks, in zip archives and in datalink layer protocols. See http://docs.python.org/library/binascii.html for a python module that contains the 32 bits CRC
   - the Alder checksum defined in **RFC 2920** for the SCTP protocol but replaced by a CRC later **RFC 3309**
   - the Fletcher checksum [Fletcher1982], see http://drdobbs.com/database/184408761 for implementation details

   By using your knowledge of the Internet checksum, can you find a transmission error that will not be detected by the Internet checksum ?

4. The CRCs are efficient error detection codes that are able to detect :

   - all errors that affect an odd number of bits
   - all errors that affect a sequence of bits which is shorter than the length of the CRC

   Carry experiments with one implementation of CRC-32 to verify that this is indeed the case.

5. Checksums and CRCs should not be confused with secure hash functions such as MD5 defined in **RFC 1321** or SHA-1 described in **RFC 4634**. Secure hash functions are used to ensure that files or sometimes packets/segments have not been modified. Secure hash functions aim at detecting malicious changes while checksums and CRCs only detect random transmission errors. Perform some experiments with hash functions such as those defined in the http://docs.python.org/library/hashlib.html python hashlib module to verify that this is indeed the case.

6. A version of the Alternating Bit Protocol supporting variable length segments uses a header that contains the following fields :

   - a *number* (0 or 1)
   - a *length* field that indicates the length of the data

- a *CRC*

To speedup the transmission of the segments, a student proposes to compute the CRC over the data part of the segment but not over the header. What do you think of this optimisation ?

7. On Unix hosts, the `ping(8)` command can be used to measure the round-trip-time to send and receive packets from a remote host. Use `ping(8)` to measure the round-trip to a remote host. Chose a remote destination which is far from your current location, e.g. a small web server in a distant country. There are implementations of ping in various languages, see e.g. http://pypi.python.org/pypi/ping/0.2 for a python implementation of ''ping''
8. How would you set the retransmission timer if you were implementing the Alternating Bit Protocol to exchange files with a server such as the one that you measured above ?
9. What are the factors that affect the performance of the Alternating Bit Protocol ?
10. Links are often considered as symmetrical, i.e. they offer the same bandwidth in both directions. Symmetrical links are widely used in Local Area Networks and in the core of the Internet, but there are many asymmetrical link technologies. The most common example are the various types of ADSL and CATV technologies. Consider an implementation of the Alternating Bit Protocol that is used between two hosts that are directly connected by using an asymmetric link. Assume that a host is sending segments containing 10 bytes of control information and 90 bytes of data and that the acknowledgements are 10 bytes long. If the round-trip-time is negligible, what is the minimum bandwidth required on the return link to ensure that the transmission of acknowledgements is not a bottleneck ?
11. Derive a mathematical expression that provides the *goodput* achieved by the Alternating Bit Protocol assuming that :
    - Each segment contains *D* bytes of data and *c* bytes of control information
    - Each acknowledgement contains *c* bytes of control information
    - The bandwidth of the two directions of the link is set to *B* bits per second
    - The delay between the two hosts is *s* seconds in both directions

    The *goodput* is defined as the amount of SDUs (measured in bytes) that is successfully transferred during a period of time
12. Consider an Alternating Bit Protocol that is used over a link that suffers from deterministic errors. When the error ratio is set to $\frac{1}{p}$, this means that $p-1$ bits are transmitted correctly and the p^{th} bit is corrupted. Discuss the factors that affect the performance of the Alternating Bit Protocol over such a link.
13. Amazon provides the S3 storage service where companies and researchers can store lots of information and perform computations on the stored information. Amazon allows users to send files through the Internet, but also by sending hard-disks. Assume that a 1 Terabyte hard-disk can be delivered within 24 hours to Amazon by courier service. What is the minimum bandwidth required to match the bandwidth of this courier service ?
14. Several large data centers operators (e.g. Microsoft and google) have announced that they install servers as containers with each container hosting up to 2000 servers. Assuming a container with 2000 servers and each storing 500 GBytes of data, what is the time required to move all the data stored in one container over one 10 Gbps link ? What is the bandwidth of a truck that needs 10 hours to move one container from one data center to another.
15. What are the techniques used by a go-back-n sender to recover from :
    - transmission errors
    - losses of data segments
    - losses of acknowledgements
16. Consider a *b* bits per second link between two hosts that has a propagation delay of *t* seconds. Derive a formula that computes the time elapsed between the transmission of the first bit of a *d* bytes segment from a sending host and the reception of the last bit of this segment on the receiving host.

17. Consider a go-back-n sender and a go-back receiver that are directly connected with a 10 Mbps link that has a propagation delay of 100 milliseconds. Assume that the retransmission timer is set to three seconds. If the window has a length of 4 segments, draw a time-sequence diagram showing the transmission of 10 segments (each segment contains 10000 bits):
    - when there are no losses
    - when the third and seventh segments are lost
    - when the second, fourth, sixth, eighth, ... acknowledgements are lost
    - when the third and fourth data segments are reordered (i.e. the fourth arrives before the third)
18. Same question when using selective repeat instead of go-back-n. Note that the answer is not necessarily the same.
19. Consider two high-end servers connected back-to-back by using a 10 Gbps interface. If the delay between the two servers is one millisecond, what is the throughput that can be achieved by a transport protocol that is using 10,000 bits segments and a window of
    - one segment
    - ten segments
    - hundred segments
20. Consider two servers are directly connected by using a *b* bits per second link with a round-trip-time of *r* seconds. The two servers are using a transport protocol that sends segments containing *s* bytes and acknowledgements composed of *a* bytes. Can you derive a formula that computes the smallest window (measured in segments) that is required to ensure that the servers will be able to completely utilise the link ?
21. Same question as above if the two servers are connected through an asymmetrical link that transmits *bu* bits per second in the direction used to send data segments and *bd* bits per second in the direction used to send acknowledgements.
22. The Trivial File Transfer Protocol is a very simple file transfer protocol that is often used by disk-less hosts when booting from a server. Read the TFTP specification in **RFC 1350** and explain how TFTP recovers from transmission errors and losses.
23. Is it possible for a go-back-n receiver to inter-operate with a selective-repeat sender ? Justify your answer.
24. Is it possible for a selective-repeat receiver to inter-operate with a go-back-n sender ? Justify your answer.
25. The go-back-n and selective repeat mechanisms that are described in the book exclusively rely on cumulative acknowledgements. This implies that a receiver always returns to the sender information about the last segment that was received in-sequence. If there are frequent losses or reordering, a selective repeat receiver could return several times the same cumulative acknowledgment. Can you think of other types of acknowledgements that could be used by a selective repeat receiver to provide additional information about the out-of-sequence segments that it has received. Design such acknowledgements and explain how the sender should react upon reception of this information.
26. The *goodput* achieved by a transport protocol is usually defined as the number of application layer bytes that are exchanged per unit of time. What are the factors that can influence the *goodput* achieved by a given transport protocol ?
27. When used with IPv4, Transmission Control Protocol (TCP) attaches 40 bytes of control information to each segment sent. Assuming an infinite window and no losses nor transmission errors, derive a formula that computes the maximum TCP goodput in function of the size of the segments that are sent.
28. A go-back-n sender uses a window size encoded in a *n* bits field. How many segments can it send without receiving an acknowledgement ?
29. Consider the following situation. A go-back-n receiver has sent a full window of data segments. All the segments have been received correctly and in-order by the receiver, but all the returned acknowledgements have been lost. Show by using a time sequence diagram (e.g. by considering a window of four segments) what happens in this case. Can you fix the problem on the go-back-n sender ?

30. Same question as above, but assume now that both the sender and the receiver implement selective repeat. Note the the answer will be different from the above question.

31. Consider a transport that supports window of one hundred 1250 Bytes segments. What is the maximum bandwidth that this protocol can achieve if the round-trip-time is set to one second ? What happens if, instead of advertising a window of one hundred segments, the receiver decides to advertise a window of 10 segments ?

32. Explain under which circumstances a transport entity could advertise a window of 0 segments ?

33. To understand the operation of the TCP congestion control mechanism, it is useful to draw some time sequence diagrams. Let us consider a simple scenario of a web client connected to the Internet that wishes to retrieve a simple web page from a remote web server. For simplicity, we will assume that the delay between the client and the server is 0.5 seconds and that the packet transmission times on the client and the servers are negligible (e.g. they are both connected to a 1 Gbps network). We will also assume that the client and the server use 1 KBytes segments.

    1. Compute the time required to open a TCP connection, send an HTTP request and retrieve a 16 KBytes web page. This page size is typical of the results returned by search engines like **google_** or bing. An important factor in this delay is the initial size of the TCP congestion window on the server. Assume first that the initial window is set to 1 segment as defined in **RFC 2001**, 4 KBytes (i.e. 4 segments in this case) as proposed in **RFC 3390** or 16 KBytes as proposed in a recent paper.
    2. Perform the same analysis with an initial window of one segment is the third segment sent by the server is lost and the retransmission timeout is fixed and set to 2 seconds.
    3. Same question as above but assume now that the 6th segment is lost.
    4. Same question as above, but consider now the loss of the second and seventh acknowledgements sent by the client.
    5. Does the analysis above changes if the initial window is set to 16 KBytes instead of one segment ?

34. Several MBytes have been sent on a TCP connection and it becomes idle for several minutes. Discuss which values should be used for the congestion window, slow start threshold and retransmission timers.

35. To operate reliably, a transport protocol that uses Go-back-n (resp. selective repeat) cannot use a window that is larger than $2^n - 1$ (resp. 2^{n-1}) segments. Does this limitation affects TCP ? Explain your answer.

36. Consider the simple network shown in the figure below. In this network, the router between the client and the server can only store on each outgoing interface one packet in addition to the packet that it is currently transmitting. It discards all the packets that arrive while its buffer is full. Assuming that you can neglect the transmission time of acknowledgements and that the server uses an initial window of one segment and has a retransmission timer set to 500 milliseconds, what is the time required to transmit 10 segments from the client to the server. Does the performance increases if the server uses an initial window of 16 segments instead ?

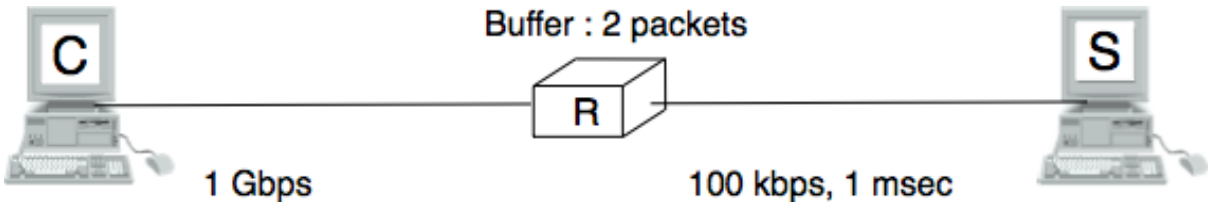

Figure 4.60: Simple network

37. The figure below describes the evolution of the congestion window of a TCP connection. Can you find the reasons for the three events that are marked in the figure ?

38. The figure below describes the evolution of the congestion window of a TCP connection. Can you find the reasons for the three events that are marked in the figure ?

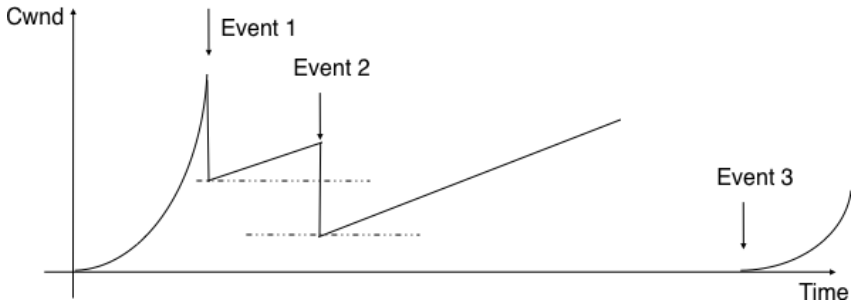

Figure 4.61: Evolution of the congestion window

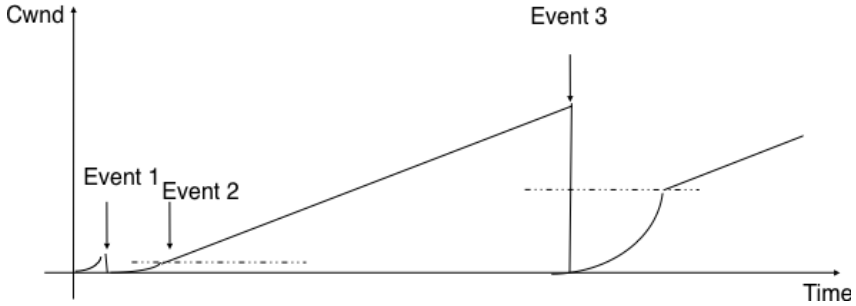

Figure 4.62: Evolution of the congestion window

39. A web server serves mainly HTML pages that fit inside 10 TCP segments. Assuming that the transmission time of each segment can be neglected, compute the total transfer time of such a page (in round-trip-times) assuming that :

- the TCP stack uses an initial window size of 1 segment
- the TCP stack uses an initial window size of three segments

40. **RFC 3168** defines mechanism that allow routers to mark packets by setting one bit in the packet header when they are congested. When a TCP destination receives such a marking in a packet, it returns the congestion marking to the source that reacts by halving its congestion window and performs congestion avoidance. Consider a TCP connection where the fourth data segment experiences congestion. Compare the delay to transmit 8 segments in a network where routers discards packets during congestion and a network where routers mark packets during congestion.

4.5.2 Practice

1. The socket interface allows you to use the UDP protocol on a Unix host. UDP provides a connectionless unreliable service that in theory allows you to send SDUs of up to 64 KBytes.

- Implement a small UDP client and a small UDP server (in python, you can start from the example provided in http://docs.python.org/library/socket.html but you can also use C or java)
- run the client and the servers on different workstations to determine experimentally the largest SDU that is supported by your language and OS. If possible, use different languages and Operating Systems in each group.

2. By using the socket interface, implement on top of the connectionless unreliable service provided by UDP a simple client that sends the following message shown in the figure below.

   In this message, the bit flags should be set to *01010011b*, the value of the 16 bits field must be the square root of the value contained in the 32 bits field, the character string must be an ASCII representation (without any trailing *0*) of the number contained in the 32 bits character field. The last 16 bits of the message contain an Internet checksum that has been computed over the entire message.

   Upon reception of a message, the server verifies that :

- the flag has the correct value
- the 32 bits integer is the square of the 16 bits integer
- the character string is an ASCII representation of the 32 bits integer
- the Internet checksum is correct

If the verification succeeds, the server returns a SDU containing *11111111b*. Otherwise it returns *01010101b*

Your implementation must be able to run on both low endian and big endian machines. If you have access to different types of machines (e.g. x86 laptops and SPARC servers), try to run your implementation on both types of machines.

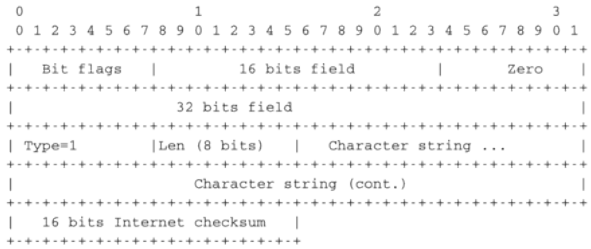

Figure 4.63: Simple SDU format

3. The socket library is also used to develop applications above the reliable bytestream service provided by TCP. We have installed on the *cnp3.info.ucl.ac.be* server a simple server that provides a simple client-server service. The service operates as follows :

   - the server listens on port *62141* for a TCP connection
   - upon the establishment of a TCP connection, the server sends an integer by using the following TLV format :
     - the first two bits indicate the type of information (01 for ASCII, 10 for boolean)
     - the next six bits indicate the length of the information (in bytes)
     - An ASCII TLV has a variable length and the next bytes contain one ASCII character per byte. A boolean TLV has a length of one byte. The byte is set to *00000000b* for *true* and *00000001b* for false.
   - the client replies by sending the received integer encoded as a 32 bits integer in *network byte order*
   - the server returns a TLV containing *true* if the integer was correct and a TLV containing *false* otherwise and closes the TCP connection

   Implement a client to interact with this server in C, Java or python.

4. It is now time to implement a small transport protocol. The protocol uses a sliding window to transmit more than one segment without being forced to wait for an acknowledgment. Your implementation must support variable size sliding window as the other end of the flow can send its maximum window size. The window size is encoded as a three bits unsigned integer.

   The protocol identifies the DATA segments by using sequence numbers. The sequence number of the first segment must be 0. It is incremented by one for each new segment. The receiver must acknowledge the delivered segments by sending an ACK segment. The sequence number field in the ACK segment always contains the sequence number of the next expected in-sequence segment at the receiver. The flow of data is unidirectional, meaning that the sender only sends DATA segments and the receiver only sends ACK segments.

   To deal with segments losses, the protocol must implement a recovery technique such as go-back-n or selective repeat and use retransmission timers. You can select the technique that best suite your needs and start from a simple technique that you improve later.

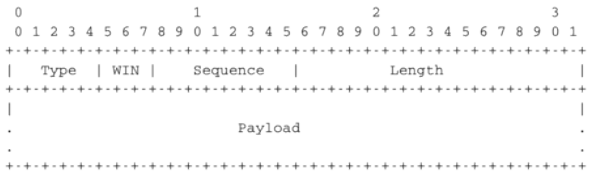

Figure 4.64: Segment format

This segment format contains the following fields :

- *Type*: segment type
  - 0x1 DATA segment.
  - 0x2 ACK segment
- *WIN*: the size of the current window (an integer encoded as a 3 bits field). In DATA segments, this field indicates the size of the sending window of the sender. In ACK segments, this field indicates the current value of the receiving window.
- *Sequence*: Sequence number (8 bits unsigned integer), starts at 0. The sequence number is incremented by 1 for each new DATA segment sent by the sender. Inside an ACK segment, the sequence field carries the sequence number of the next in-sequence segment that is expected by the receiver.
- *Length*: length of the payload in multiple of one byte. All DATA segments contain a payload with 512 bytes of data, except the last DATA segment of a transfer that can be shorter. The reception of a DATA segment whose length is different than 512 indicates the end of the data transit.
- *Payload*: the data to send

The client and the server exchange UDP datagrams that contain the DATA and ACK segments. They must provide a command-line interface that allows to transmit one binary file and support the following parameters :

```
sender <destination_DNS_name> <destination_port_number> <window_size> <input_file>
receiver <listening_port_number> <window_size> <output_file>
```

In order to test the reactions of your protocol against errors and losses, you you can use a random number generator to probabilistically drop received segments and introduce random delays upon the arrival of a segment.

Packet trace analysis

When debugging networking problems or to analyse performance problems, it is sometimes useful to capture the segments that are exchanged between two hosts and to analyse them.

Several packet trace analysis tools are available, either as commercial or open-source tools. These tools are able to capture all the packets exchanged on a link. Of course, capturing packets require administrator privileges. They can also analyse the content of the captured packets and display information about them. The captured packets can be stored in a file for offline analysis.

tcpdump is probably one of the most well known packet capture software. It is able to both capture packets and display their content. tcpdump is a text-based tool that can display the value of the most important fields of the captured packets. Additional information about tcpdump may be found in *tcpdump(1)*. The text below is an example of the output of tcpdump for the first TCP segments exchanged on an scp transfer between two hosts.

```
21:05:56.230737 IP 192.168.1.101.54150 > 130.104.78.8.22: S 1385328972:1385328972(0) win 65535 <ms
21:05:56.251468 IP 130.104.78.8.22 > 192.168.1.101.54150: S 3627767479:3627767479(0) ack 13853289
21:05:56.251560 IP 192.168.1.101.54150 > 130.104.78.8.22: . ack 1 win 65535 <nop,nop,timestamp 27
21:05:56.279137 IP 130.104.78.8.22 > 192.168.1.101.54150: P 1:21(20) ack 1 win 49248 <nop,nop,time
```

```
21:05:56.279241 IP 192.168.1.101.54150 > 130.104.78.8.22: . ack 21 win 65535 <nop,nop,timestamp 2
21:05:56.279534 IP 192.168.1.101.54150 > 130.104.78.8.22: P 1:22(21) ack 21 win  65535 <nop,nop,t
21:05:56.303527 IP 130.104.78.8.22 > 192.168.1.101.54150: . ack 22 win 49248 <nop,nop,timestamp 1
21:05:56.303623 IP 192.168.1.101.54150 > 130.104.78.8.22: P 22:814(792) ack 21 win 65535 <nop,nop
```

You can easily recognise in the output above the *SYN* segment containing the *MSS*, *window scale*, *timestamp* and *sackOK* options, the *SYN+ACK* segment whose *wscale* option indicates that the server uses window scaling for this connection and then the first few segments exchanged on the connection.

wireshark is more recent than tcpdump. It evolved from the ethereal packet trace analysis software. It can be used as a text tool like tcpdump. For a TCP connection, wireshark would provide almost the same output as tcpdump. The main advantage of wireshark is that it also includes a graphical user interface that allows to perform various types of analysis on a packet trace.

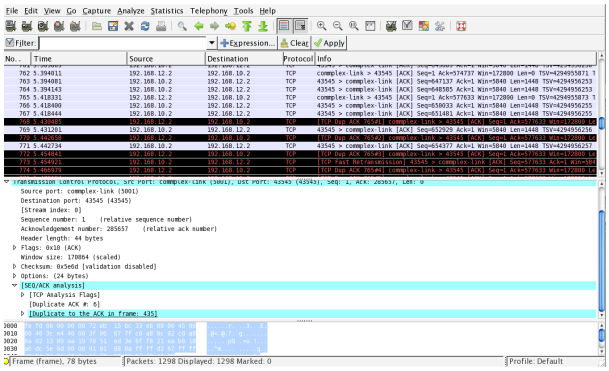

Figure 4.65: Wireshark : default window

The wireshark window is divided in three parts. The top part of the window is a summary of the first packets from the trace. By clicking on one of the lines, you can show the detailed content of this packet in the middle part of the window. The middle of the window allows you to inspect all the fields of the captured packet. The bottom part of the window is the hexadecimal representation of the packet, with the field selected in the middle window being highlighted.

wireshark is very good at displaying packets, but it also contains several analysis tools that can be very useful. The first tool is *Follow TCP stream*. It is part of the *Analyze* menu and allows you to reassemble and display all the payload exchanged during a TCP connection. This tool can be useful if you need to analyse for example the commands exchanged during a SMTP session.

The second tool is the flow graph that is part of the *Statistics* menu. It provides a time sequence diagram of the packets exchanged with some comments about the packet contents. See blow for an example.

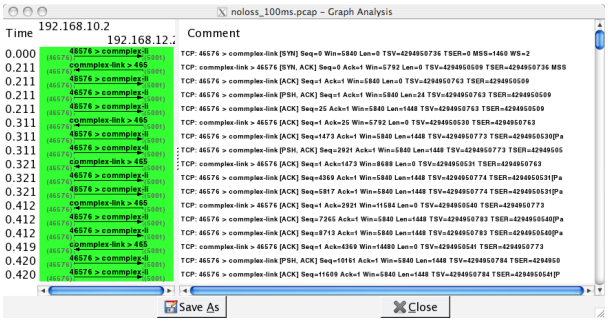

Figure 4.66: Wireshark : flow graph

The third set of tools are the *TCP stream graph* tools that are part of the *Statistics menu*. These tools allow you to plot various types of information extracted from the segments exchanged during a TCP connection. A first interesting graph is the *sequence number graph* that shows the evolution of the sequence number field of the captured segments with time. This graph can be used to detect graphically retransmissions.

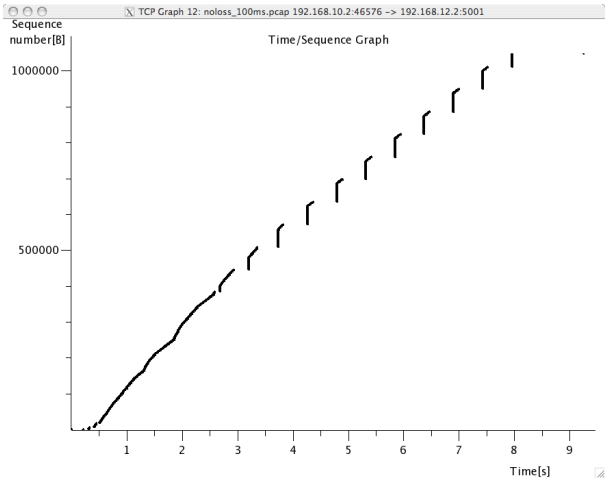

Figure 4.67: Wireshark : sequence number graph

A second interesting graph is the *round-trip-time* graph that shows the evolution of the round-trip-time in function of time. This graph can be used to check whether the round-trip-time remains stable or not. Note that from a packet trace, wireshark can plot two *round-trip-time* graphs, One for the flow from the client to the server and the other one. wireshark will plot the *round-trip-time* graph that corresponds to the selected packet in the top wireshark window.

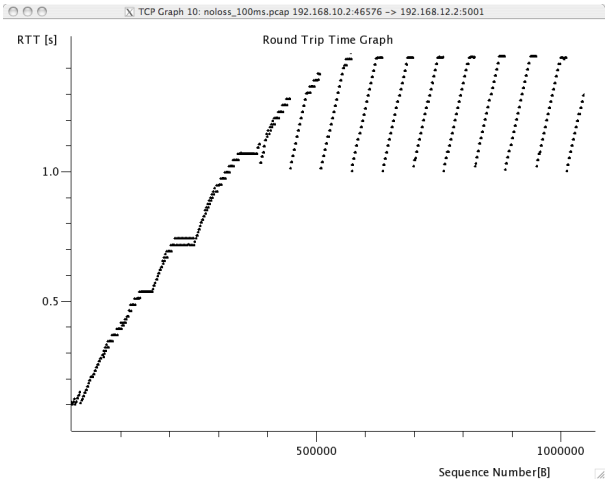

Figure 4.68: Wireshark : round-trip-time graph

Emulating a network with netkit

Netkit is network emulator based on User Mode Linux. It allows to easily set up an emulated network of Linux machines, that can act as end-host or routers.

Note: Where can I find Netkit?

Netkit is available at http://www.netkit.org. Files can be downloaded from http://wiki.netkit.org/index.php/Download_Official, and instructions for the installations are available here : http://wiki.netkit.org/download/netkit/INSTALL .

There are two ways to use Netkit : The manual way, and by using pre-configured labs. In the first case, you boot and control each machine individually, using the commands starting with a "v" (for virtual machine). In the second case, you can start a whole network in a single operation. The commands for controlling the lab start with a "l". The man pages of those commands is available from http://wiki.netkit.org/man/man7/netkit.7.html

You must be careful not to forgot to stop your virtual machines and labs, using either *vhalt* or *lhalt*.

A netkit lab is simply a directory containing at least a configuration file called *lab.conf*, and one directory for each virtual machine. In the case the lab available on iCampus, the network is composed of two pcs, *pc1* and *pc2*, both of them being connected to a router *r1*. The lab.conf file contains the following lines :

```
pc1[0]=A
pc2[0]=B
r1[0]=A
r1[1]=B
```

This means that *pc1* and *r1* are connected to a "virtual LAN" named *A* via their interface *eth0*, while *pc2* and *r1* are connected to the "virtual LAN" *B* via respectively their interfaces *eth0* and *eth1*.

The directory of each device is initially empty, but will be used by Netkit to store their filesystem.

The lab directory can contain optional files. In the lab provided to you, the "pc1.startup" file contains the shell instructions to be executed on startup of the virtual machine. In this specific case, the script configures the interface *eth0* to allow traffic exchanges between *pc1* and *r1*, as well as the routing table entry to join *pc2*.

Starting a lab consists thus simply in unpacking the provided archive, going into the lab directory and typing *lstart* to start the network.

Note: File sharing between virtual machines and host

Virtual machines can access to the directory of the lab they belong to. This repertory is mounted in their filesystem at the path */hostlab*.

In the netkit lab (`exercises/netkit/netkit_lab_2hosts_1rtr_ipv4.tar.tar.gz`, you can find a simple python client/server application that establishes TCP connections. Feel free to re-use this code to perform your analysis.

Note: netkit tools

As the virtual machines run Linux, standard networking tools such as hping, tcpdump, netstat etc. are available as usual.

Note that capturing network traces can be facilitated by using the *uml_dump* extension available at http://kartoch.msi.unilim.fr/blog/?p=19 . This extension is already installed in the Netkit installation on the student lab. In order to capture the traffic exchanged on a given 'virtual LAN', you simply need to issue the command *vdump <LAN name>* on the host. If you want to pipe the trace to wireshark, you can use *vdump A | wireshark -i - -k*

---

1. A TCP/IP stack receives a SYN segment with the sequence number set to 1234. What will be the value of the acknowledgement number in the returned SYN+ACK segment ?
2. Is it possible for a TCP/IP stack to return a SYN+ACK segment with the acknowledgement number set to *0* ? If no, explain why. If yes, what was the content of the received SYN segment.
3. Open the tcpdump packet trace `exercises/traces/trace.5connections_opening_closing.pcap` and identify the number of different TCP connections that are established and closed. For each connection, explain by which mechanism they are closed. Analyse the initial sequence numbers that are used in the SYN and SYN+ACK segments. How do these initial sequence numbers evolve ? Are they increased every 4 microseconds ?
4. The tcpdump packet trace `exercises/traces/trace.5connections.pcap` contains several connection attempts. Can you explain what is happening with these connection attempts ?
5. The tcpdump packet trace `exercises/traces/trace.ipv6.google.com.pcap` was collected from a popular website that is accessible by using IPv6. Explain the TCP options that are supported by the client and the server.

6. The tcpdump packet trace `exercises/traces/trace.sirius.info.ucl.ac.be.pcap` Was collected on the departmental server. What are the TCP options supported by this server ?

7. A TCP implementation maintains a Transmission Control Block (TCB) for each TCP connection. This TCB is a data structure that contains the complete "*state*" of each TCP connection. The TCB is described in **RFC 793**. It contains first the identification of the TCP connection :

   - *localip* : the IP address of the local host
   - *remoteip* : the IP address of the remote host
   - *remoteport* : the TCP port used for this connection on the remote host
   - *localport* : the TCP port used for this connection on the local host. Note that when a client opens a TCP connection, the local port will often be chosen in the ephemeral port range (49152 <= localport <= 65535).
   - *sndnxt* : the sequence number of the next byte in the byte stream (the first byte of a new data segment that you send will use this sequence number)
   - *snduna* : the earliest sequence number that has been sent but has not yet been acknowledged
   - *rcvnxt* : the sequence number of the next byte that your implementation expects to receive from the remote host. For this exercise, you do not need to maintain a receive buffer and your implementation can discard the out-of-sequence segments that it receives
   - *sndwnd* : the current sending window
   - *rcvwnd* : the current window advertised by the receiver

   Using the `exercises/traces/trace.sirius.info.ucl.ac.be.pcap` packet trace, what is the TCB of the connection on host *130.104.78.8* when it sends the third segment of the trace ?

8. The tcpdump packet trace `exercises/traces/trace.maps.google.com` was collected by containing a popular web site that provides mapping information. How many TCP connections were used to retrieve the information from this server ?

9. Some network monitoring tools such as ntop collect all the TCP segments sent and received by a host or a group of hosts and provide interesting statistics such as the number of TCP connections, the number of bytes exchanged over each TCP connection, ... Assuming that you can capture all the TCP segments sent by a host, propose the pseudo-code of an application that would list all the TCP connections established and accepted by this host and the number of bytes exchanged over each connection. Do you need to count the number of bytes contained inside each segment to report the number of bytes exchanged over each TCP connection ?

10. There are two types of firewalls [30] : special devices that are placed at the border of campus or enterprise networks and software that runs on endhosts. Software firewalls typically analyse all the packets that are received by a host and decide based on the packet's header and contents whether it can be processed by the host's network stack or must be discarded. System administrators often configure firewalls on laptop or student machines to prevent students from installing servers on their machines. How would you design a simple firewall that blocks all incoming TCP connections but still allows the host to establish TCP connections to any remote server ?

11. Using the netkit lab explained above, perform some tests by using *hping3(8)*. *hping3(8)* is a command line tool that allows anyone (having system administrator privileges) to send special IP packets and TCP segments. *hping3(8)* can be used to verify the configuration of firewalls [33] or diagnose problems. We will use it to test the operation of the Linux TCP stack running inside netkit.

    1. On the server host, launch *tcpdump(1)* with *-vv* as parameter to collect all packets received from the client and display them. Using *hping3(8)* on the client host, send a valid SYN segment to one unused port on the server host (e.g. *12345*). What are the contents of the segment returned by the server ?

[30] A firewall is a software or hardware device that analyses TCP/IP packets and decides, based on a set of rules, to accept or discard the packets received or sent. The rules used by a firewall usually depend on the value of some fields of the packets (e.g. type of transport protocols, ports, ...). We will discuss in more details the operation of firewalls in the network layer chapter.

2. Perform the same experiment, but now send a SYN segment towards port *7*. This port is the default port for the discard service (see `services(5)`) launched by `xinetd(8)`). What segment does the server sends in reply ? What happens upon reception of this segment ? Explain your answer.

12. The Linux TCP/IP stack can be easily configured by using `sysctl(8)` to change kernel configuration variables. See http://fasterdata.es.net/TCP-tuning/ip-sysctl-2.6.txt for a recent list of the sysctl variables on the Linux TCP/IP stack. Try to disable the selective acknowledgements and the RFC1323 timestamp and large window options and open a TCP connection on port *7* on the server by using :manpage:telnet‘(1)‘. Check by using `tcpdump(1)` the effect of these kernel variables on the segments sent by the Linux stack in netkit.

13. Network administrators sometimes need to verify which networking daemons are active on a server. When logged on the server, several tools can be used to verify this. A first solution is to use the `netstat(8)` command. This command allows you to extract various statistics from the networking stack on the Linux kernel. For TCP, *netstat* can list all the active TCP connections with the state of their FSM. *netstat* supports the following options that could be useful during this exercises :

    - *-t* requests information about the TCP connections
    - *-n* requests numeric output (by default, *netstat* sends DNS queries to resolve IP addresses in hosts and uses */etc/services* to convert port number in service names, *-n* is recommended on netkit machines)
    - *-e* provides more information about the state of the TCP connections
    - *-o* provides information about the timers
    - *-a* provides information about all TCP connections, not only those in the Established state

    On the netkit lab, launch a daemon and start a TCP connection by using `telnet(1)` and use `netstat(8)` to verify the state of these connections.

    A second solution to determine which network daemons are running on a server is to use a tool like `nmap(1)`. `nmap(1)` can be run remotely and thus can provide information about a host on which the system administrator cannot login. Use `tcpdump(1)` to collect the segments sent by `nmap(1)` running on the client and explain how `nmap(1)` operates.

14. Long lived TCP connections are susceptible to the so-called *RST attacks*. Try to find additional information about this attack and explain how a TCP stack could mitigate such attacks.

15. For the exercises below, we have performed measurements in an emulated [31] network similar to the one shown below.

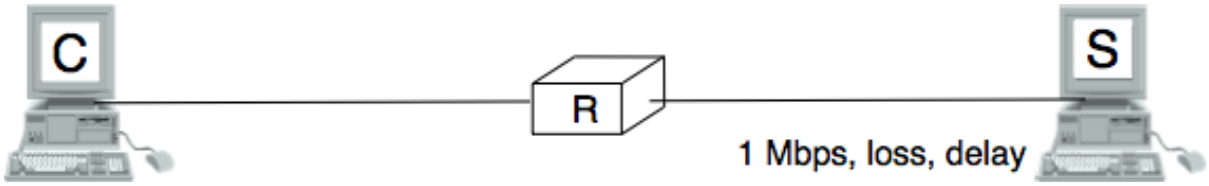

Figure 4.69: Emulated network

The emulated network is composed of three UML machines [32]: a client, a server and a router. The client and the server are connected via the router. The client sends data to the server. The link between the router and the client is controlled by using the netem Linux kernel module. This module allows us to insert additional delays, reduce the link bandwidth and insert random packet losses.

[31] With an emulated network, it is more difficult to obtain quantitative results than with a real network since all the emulated machines need to share the same CPU and memory. This creates interactions between the different emulated machines that do not happen in the real world. However, since the objective of this exercise is only to allow the students to understand the behaviour of the TCP congestion control mechanism, this is not a severe problem.

[32] For more information about the TCP congestion control schemes implemented in the Linux kernel, see http://linuxgazette.net/135/pfeiffer.html and http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf or the source code of a recent Linux. A description of some of the sysctl variables that allow to tune the TCP implementation in the Linux kernel may be found in http://fasterdata.es.net/TCP-tuning/linux.html. For this exercise, we have configured the Linux kernel to use the NewReno scheme **RFC 3782** that is very close to the official standard defined in **RFC 5681**

We used netem to collect several traces :

- `exercises/traces/trace0.pcap`
- `exercises/traces/trace1.pcap`
- `exercises/traces/trace2.pcap`
- `exercises/traces/trace3.pcap`

Using wireshark or tcpdump, carry out the following analyses :

1. Identify the TCP options that have been used on the TCP connection
2. Try to find explanations for the evolution of the round-trip-time on each of these TCP connections. For this, you can use the *round-trip-time* graph of wireshark, but be careful with their estimation as some versions of wireshark are buggy
3. Verify whether the TCP implementation used implemented *delayed acknowledgements*
4. Inside each packet trace, find :

1. one segment that has been retransmitted by using *fast retransmit*. Explain this retransmission in details.
2. one segment that has been retransmitted thanks to the expiration of TCP's retransmission timeout. Explain why this segment could not have been retransmitted by using *fast retransmit*.
5. wireshark contains several two useful graphs : the *round-trip-time* graph and the *time sequence* graph. Explain how you would compute the same graph from such a trace .
6. When displaying TCP segments, recent versions of wireshark contain *expert analysis* heuristics that indicate whether the segment has been retransmitted, whether it is a duplicate ack or whether the retransmission timeout has expired. Explain how you would implement the same heuristics as wireshark.
7. Can you find which file has been exchanged during the transfer ?

16. You have been hired as an networking expert by a company. In this company, users of a networked application complain that the network is very slow. The developers of the application argue that any delays are caused by packet losses and a buggy network. The network administrator argues that the network works perfectly and that the delays perceived by the users are caused by the applications or the servers where the application is running. To resolve the case and determine whether the problem is due to the network or the server on which the application is running. The network administrator has collected a representative packet trace that you can download from `exercises/traces/trace9.pcap`. By looking at the trace, can you resolve this case and indicate whether the network or the application is the culprit ?